University of Zurich[UZH]

Department of Informatics

Querying Databases With Ongoing Time Intervals

Dissertation submitted to the Faculty of Business,
Economics and Informatics
of the University of Zurich

to obtain the degree of
Doktorin der Wissenschaften, Dr. sc.
(corresponds to Doctor of Science, PhD)

presented by

Yvonne Mülle

from Karlsruhe, Germany

approved in February 2021

at the request of

Prof. Dr. Michael H. Böhlen
Prof. Dr. Techn. Christian S. Jensen

University of Zurich[UZH]

The Faculty of Business, Economics and Informatics of the University of Zurich hereby authorizes the printing of this dissertation, without indicating an opinion of the views expressed in the work.

Zurich, February 17, 2021

The Chairman of the Doctoral Board: Prof. Dr. Thomas Fritz

Abstract

Data that are associated with a valid time interval are present in many real-world applications that deal with employment contracts, insurance policies, software bugs, etc. The ongoing time point *now* is commonly used in these applications to state that the contract, policy, software bug is valid from the start point onward. Data with ongoing time points have far-reaching consequences for database systems since ongoing time points change continuously as time passes by. The approaches so far deal with ongoing time points by instantiating them to the reference time. This yields query results that are only valid at the chosen time and get invalidated by time passing by. Since query results do not contain ongoing time points anymore, it is impossible for applications to identify valid time intervals that change as time passes by. The goal of this thesis is to evaluate queries at every possible reference time to get query results that remain valid as time passes by and to find result representations in which ongoing time points are preserved.

The solution we provide supports predicates, functions, and relational algebra operators on ongoing data types. These include, but are not limited to, the most commonly used temporal predicates *overlaps*, *before*, and *during*, the three interval functions *intersection*, *difference*, and *union*, which are the standard functions and building blocks for processing time intervals, and the widely used relational algebra operators *projection*, *selection*, *inner join*, and *aggregation*.

To get results that remain valid, we keep ongoing time points *uninstantiated* during query processing. At each reference time, the result of a predicate, function, or relational algebra operator on ongoing data types is equal to the result obtained by evaluating the corresponding predicate,

function, or relational algebra operator for fixed data types on the instantiated input arguments. The fixed data types correspond to the ongoing data types but do not contain ongoing values.

We represent the results as a combination of ongoing values and the reference times when these values are part of the result. At each reference time, the result contains ongoing values that represent the result at this reference time. Predicates change their truth value, i.e., *true* and *false*, depending on the reference time. We associate each truth value with the reference times when the predicate has this truth value. Interval functions on ongoing time intervals evaluate to different time intervals depending on the reference time. We represent the function result as pairs of ongoing intervals and the reference times when the interval is part of the result. The results of relational algebra operators are *ongoing relations* that associate each tuple with a reference time attribute RT. The value of the RT attribute includes the reference times when *now* can be instantiated in the tuple and the tuple belongs to the instantiated relations. The RT value of a tuple is restricted by predicates and functions on ongoing attributes.

We provide an efficient implementation of the ongoing data types and the predicates, functions, and relational algebra operators on ongoing data types in the kernel of the open source database system PostgreSQL. Our integration can leverage existing database optimization strategies and algorithms, which were designed for relations without ongoing values, for evaluating queries on relations with ongoing values to results that remain valid as time passes by.

To the love of my life

Acknowledgments

I would like to thank my advisor, Prof. Michael Böhlen, for giving me the opportunity to pursue a PhD in the Database Technology Group (DBTG). I am grateful for the numerous discussions, his constructive criticism, his motivation to always strive for the better, and the countless hours he invested to help me assemble and improve my research. Thank you for guiding me through the PhD.

A special thanks to Prof. Dr. Techn. Christian S. Jensen for agreeing to be the co-advisor of my PhD thesis and to Prof. Dr. Thomas Fritz for chairing my PhD thesis defense.

I would also like to thank my colleagues and former colleagues of the Database Technology Group for the constructive feedback during our meetings and the cheerful discussions during our breaks.

Last but not least, I want to express my sincere gratitude to my partner for standing by my side along the way and for always believing in me.

Yvonne Mülle
Zurich, July 2020

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

1.1 Databases With Ongoing Time Intervals

Data that are associated with a valid time interval are present in many real-world applications that deal with employment contracts, insurance policies, software bugs, etc. The ongoing time point *now* is commonly used in these applications to state that a contract, policy, software bug is valid from the start point onward.

Ongoing time point *now* changes its value as time passes by. The reference time *rt* is used to determine the value of *now*: at each reference time, ongoing time point *now* instantiates to a time point equal to the reference time. Figure 1.1 illustrates ongoing time point *now*. At reference time 03/08, ongoing time point *now* instantiates to time point 03/08, at reference time 03/09, *now* instantiates to time point 03/09, and so on. Throughout the thesis, we use time points in the mm/dd format relative to 2020: time point 03/08 denotes March 8, 2020.

The importance of supporting ongoing time point *now* in database systems has been acknowledged by the SQL standard [MS93]: it includes the reserved keywords CURRENT_TIME, CUR-

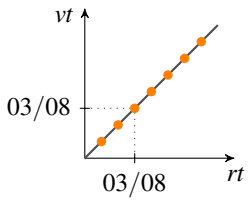


Figure 1.1: Illustration of ongoing time point *now*.

RENT_DATE, and CURRENT_TIMESTAMP that denote the ongoing time point *now* for different time granularities. These keywords can then be used in SQL queries.

Data with ongoing time points and evaluating queries on these data have far-reaching consequences for database systems. A key assumption of database systems is that query results only change after explicit data modifications, i.e., when data are inserted, updated, or deleted. This assumption no longer holds when ongoing time points are present in the database since ongoing time points change their value as time passes by. In this case, query results get also outdated by time passing by.

Example 1. *Consider a consulting company that runs software development projects. An employee has a fixed-term or a permanent employment. Fixed-term employments have fixed start points that indicate the start of the employment and fixed end dates that indicate the end of the employment. Permanent employments have fixed start dates but end dates that keep increasing until the contract is modified. These end dates are* ongoing. *Employees get assigned to software development projects. Projects have different employment levels as requirement; only employees that have exactly the required employment level can be assigned to the project. An assignment is possible for the whole timeframe of the project or only parts of it. Selected relations of our running example are shown in Figure 1.2 and discussed below.*

E

	Name	Role	Level	VT
e_1	Ann	SWE	4	$[03/08, now)$
e_2	Bob	SWE	6	$[07/20, now)$
e_3	Eve	SRE	6	$[07/20, 10/29)$
e_4	John	SRE	4	$[03/08, 08/23)$

P

	PID	LR	VT
p_1	500	4	$[02/04, 07/20)$
p_2	501	6	$[05/14, now)$

Figure 1.2: Relations with ongoing time intervals.

Relation **E** *lists selected employees. An employee is described by her* name*, her* role*, her employment* level*, and the valid time VT when she is employed. For instance, tuple* e_3 *records employee*

Eve who has a fixed-term employment. Eve is employed as site reliability engineer (SRE) on level 6 from the fixed start point $07/20$ *until the fixed end point* $10/29$ *exclusively. Tuple* e_1 *records employee Ann who has a permanent employment. Ann is employed as software engineer (SWE) on level 4 from* $03/08$ *until now. At reference time* $04/01$*, the intuitive meaning of tuple* e_1 *is that Ann is employed as SWE on level 4 from* $03/08$ *until* $04/01$ *exclusively; at reference time* $08/01$*, its meaning is that Ann is employed as SWE on level 4 from* $03/08$ *until* $08/01$ *exclusively; and so on.*

Relation **P** *lists selected software development projects. A project is described by the project id* PID*, the required employment level* LR*, and the valid time when the project is conducted. For instance, tuple* p_2 *records that the project with id 501 requires employment level 6 and that the project is conducted from* $05/14$ *until now.*

Note that both relations contain tuples with ongoing time point now in the valid time. Using ongoing time point now has the key property that the data stored in a relation remain valid as time passes by and the relation does not have to be constantly updated as time passes by to reflect ongoing dates.

To fill open head count for their projects, managers are interested in the following information:

1. *The managers want to determine the assignment timeframe for the employees that are eligible to work on their projects, i.e., employees whose employment timeframe overlaps with the timeframe of the project and who have exactly the required employment level. Query* Q_1 *retrieves this information:*

$$Q_1 = V_1 \leftarrow \pi_{PID,Name,Role,\mathbf{P}.VT \cap \mathbf{E}.VT}(\mathbf{P} \bowtie_{LR=Level \wedge \mathbf{P}.VT \text{ overlaps } \mathbf{E}.VT} \mathbf{E})$$

2. *The managers want to determine for each possible project assignment the timeframe of the project that is not covered by the assignment. The managers then want to broaden their search by relaxing the level requirement of the project to find employees for the not-covered project timeframe. Query* Q_2 *retrieves this information:*

$$Q_2 = V_2 \leftarrow \pi_{PID,LR,Name,Role,Level,Diff \cap \mathbf{E}.VT}(\pi_{PID,LR,(\mathbf{P}.VT \,-\, \mathbf{E}.VT)/Diff}(\mathbf{P} \bowtie_{LR=Level} \mathbf{E}) \bowtie_{LR < Level \wedge Diff \text{ overlaps } \mathbf{E}.VT} \mathbf{E})$$

   We write $\pi_{X/C}$ *to rename* X *to* C*.*

Observe that both queries use predicates and functions on the ongoing valid time VT. Query Q_2 applies predicate overlaps *to the result ongoing intervals of the difference function in join condition* $(LR < Level \wedge Diff$ overlaps $\mathbf{E}.VT)$.

Since queries Q_1 and Q_2 are evaluated on relations with ongoing valid time intervals, their results change as time passes by. Figure 1.3 illustrates this for query Q_1 at the two reference times $04/01$ *and* $08/01$. *The query results differ in the tuples that belong to the result relation and the attribute values.*

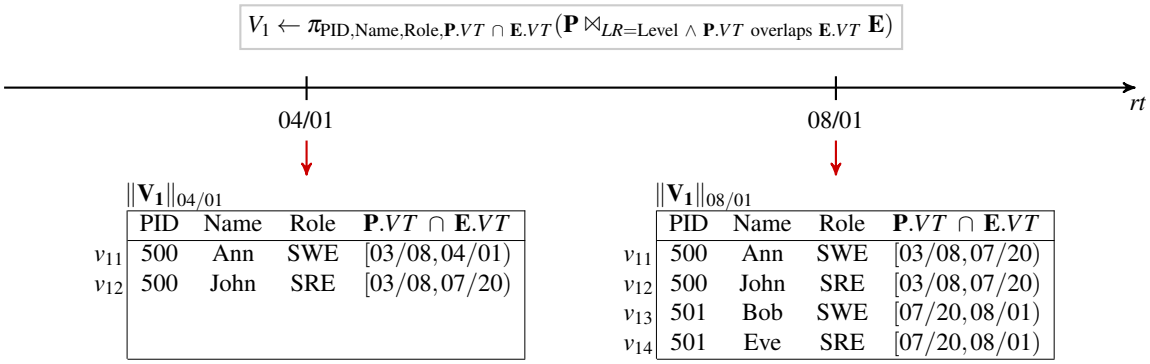


Figure 1.3: Query results get outdated by time passing by.

The query result $\|\mathbf{V_1}\|_{04/01}$ at reference time $04/01$ *includes two tuples, v_{11} and v_{12}, and the query result $\|\mathbf{V_1}\|_{08/01}$ at reference time* $08/01$ *includes four tuples, v_{11}, v_{12}, v_{13}, and v_{14}. The reason is that the truth value of the join predicate $\theta = (LR = Level \wedge \mathbf{P}.VT$ overlaps $\mathbf{E}.VT)$ changes as time passes by since the input valid times are ongoing and change as time passes by. Different tuples satisfy the join predicate at different reference times.*

The intersection $p_1.VT \cap e_1.VT$ in result tuple v_{11} changes from result interval $[03/08,04/01)$ *at reference time* $04/01$ *to result interval* $[03/08,07/20)$ *at reference time* $08/01$. *The reason is again the ongoing valid times that change as time passes by. A query result determined at one reference time gets invalidated by time passing by.*

Current database systems like PostgreSQL or Oracle cannot store ongoing time points. Instead, they instantiate the ongoing time points immediately at compile time, i.e., they replace all occurrences of ongoing time point *now* with the time when the query is processed. Various research approaches [CDI+97, TJS04, ASTS13] have progressed the basic solution offered by commercial database systems. The key idea is to store ongoing time points uninstantiated and instantiate them when accessing the data during query processing. Both ideas have in common that exist-

ing query processing techniques can be applied since the instantiation eliminates ongoing time points.

Instantiating ongoing time points before or during query processing has significant drawbacks. Torp et al. [TJS04] have shown that performing temporal modifications on tuples that are instantiated when accessed leads to incorrect modifications and thus, incorrect data in the database. In the following, we discuss significant drawbacks for query processing in general.

First, query results, including materialized views and cached query results, must be re-computed before they can be accessed. As shown in Figure 1.3, a query result computed at one reference time does not remain the correct query result for other reference times. Thus, query results always need to be re-computed to ensure that the query results that are valid at the reference time when accessed are returned.

Second, because ongoing time points are replaced by fixed time points, it is impossible for applications to identify time intervals that change as time passes by. As an example, the project assignment timeframes $\mathbf{P}.VT \cap \mathbf{E}.VT$ of query Q_1 in Example 1 are ongoing time intervals (cf. tuple v_{11} in Figure 1.3 at different reference times). The result of query Q_1 evaluated at reference time $04/01$ does not contain ongoing time points anymore. Ongoing time point *now* is replaced with the reference time $04/01$ in the time interval $p_1.VT \cap e_1.VT$ of tuple v_{11} in the query result (cf. Figure 1.3). Since the query result consists of fixed values only, it is impossible for an application to know that the intersection time interval of tuple v_{11} is ongoing and changes as time passes by.

This thesis is about elegant and efficient solutions that preserve ongoing time points in query results and that evaluate predicates, functions, and relational algebra operators at all possible reference times to get results that remain valid as time passes by. The solutions are implemented in the open source database system PostgreSQL to leave the correct handling of data with ongoing time points to the database system instead of forcing this task onto the applications that use the data.

1.2 Challenges

The two key challenges that all our approaches have in common are (1) the evaluation of operations to results that remain valid as time passes by, and (2) the finite result representation that preserves ongoing values.

Formally, given a database D with ongoing time points and a query Q that consists of relational algebra operators with predicates and functions, we want to compute a query result $Q(D)$, such that at every possible reference time rt, the query result is equivalent to the result obtained by first instantiating *now* in D and then evaluating the query on the instantiated database:

$$\forall rt\ (\|Q(D)\|_{rt} \equiv Q(\|D\|_{rt}))$$

The bind operator $\|\cdot\|_{rt}$ replaces all occurrences of *now* with the reference time rt.

Predicates, functions, and relational algebra operators each bring their own flavor to the solution of the two key challenges.

Predicates on Ongoing Values. Predicates on ongoing values change their truth value, i.e., *true* or *false*, as time passes by. As an example, predicate $\theta = (p_2.VT\ overlaps\ e_3.VT)$ is *false* up to reference time $07/20$ and it is *true* from reference time $07/21$ on:

rt	$[05/14, now)$	$[07/20, 10/29)$	$p_2.VT$ overlaps $e_3.VT$
...	...	...	...
$07/19$	$[05/14, 07/19)$	$[07/20, 10/29)$	*false*
$07/20$	$[05/14, 07/20)$	$[07/20, 10/29)$	*false*
$07/21$	$[05/14, 07/21)$	$[07/20, 10/29)$	*true*
$07/22$	$[05/14, 07/22)$	$[07/20, 10/29)$	*true*
...	...	...	...

Since predicates can either be *true* or *false* at a reference time, a natural choice to represent the truth values of a predicate for all reference times is to use the reference times when the predicate is *true*. At all other reference times, the predicate is then *false*. As an example, $(p_2.VT\ overlaps\ e_3.VT) = \{[07/21, \infty)\}$, i.e., the predicate is *true* from reference time $07/21$ on and *false* otherwise.

Since predicates on ongoing values are *true* at some reference times only, predicates in queries select tuples depending on the reference time: at the reference times when the predicate is *true*,

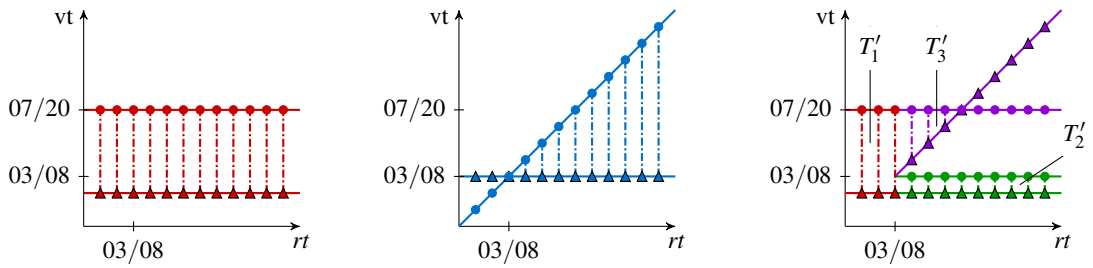
the tuple belongs to the result relation; at all other reference times, the tuple does not belong to the result relation. For instance, join predicate θ selects the joined tuple $(p_2 \circ e_3)$ at some reference times only and the joined tuple belongs to the join result from reference time $07/21$ on. To keep track of at which reference times a tuple belongs to a relation, our approach introduces *ongoing relations* that associate each tuple with a reference time attribute RT. A tuple's reference time attribute includes exactly the reference times when the tuple belongs to a relation. For instance, the result tuple derived from applying query Q_1 in Example 1 to input tuples p_2 and e_3 is the following *ongoing tuple*:

	PID	Name	Role	$\mathbf{P}.VT \cap \mathbf{E}.VT$	RT
v_{14}	501	Eve	SRE	$[07/20, +10/29)$	$\{[07/21, \infty)\}$

Ongoing time point $+10/29$ is equal to the reference time up to reference time $10/29$ and afterwards equal to time point $10/29$ (cf. Section 1.3.1). The join predicate is *true* from reference time $07/21$ on and we get $v_{14}.RT = \{[07/21, \infty)\}$. This means that tuple v_{14} belongs to the relation from reference time $07/21$ on. At the reference times that are not contained in $v_{14}.RT$, the tuple does not belong to the relation.

Functions on Ongoing Intervals. To get function results that remain valid as time passes by, we need to correctly represent the results, such that they consist of the expected time intervals at every reference time. The expected time intervals at a reference time are the time intervals one gets if all occurrences of *now* had been replaced by the reference time. As an example, for the difference function $T_1 - T_2$, the expected time intervals are the maximal sub-intervals of T_1 that do not overlap with interval T_2.

Example 2. *Consider the difference* $[02/04, 07/20) - [03/08, now)$. *The input time intervals and the difference result are illustrated in Figure 1.4.*



(a) $T_1 = [02/04, 07/20)$ (b) $T_2 = [03/08, now)$ (c) $T_1 - T_2$

Figure 1.4: The difference result with the expected time intervals at each reference time.

The result in Figure 1.4c consists of the following three pairs of ongoing interval and reference times rt:

- $T_1' = [02/04, 07/20)$ *at rt* $\in (-\infty, 03/09)$,
- $T_2' = [02/04, 03/08)$ *at rt* $\in [03/09, \infty)$, *and*
- $T_3' = [now, 07/20)$ *at rt* $\in [03/09, \infty)$.

Up to reference time $03/08$, *time interval* $[03/08, now)$ *instantiates to empty time intervals (cf. Figure 1.4b) and the input time intervals do not overlap. Thus, the difference result is the input interval* $[02/04, 07/20)$ *for reference times in* $(-\infty, 03/09)$. *From reference time* $03/09$ *on, time interval* $[03/08, now)$ *instantiates to non-empty time intervals and the input time intervals overlap. The difference result are the sub-intervals* $[02/04, 03/08)$ *and* $[now, 07/20)$ *for reference times in* $[03/09, \infty)$.

For the valid time intervals, it is important how the time intervals are represented. The reason is that we want to evaluate predicates and functions on the valid time intervals (cf. query Q_2 in Example 1) and their results depend on the representation of the input time intervals.

Example 3. *Consider query* Q_2 *in Example 1, which applies predicate* overlaps *to the difference* $\mathbf{P}.VT - \mathbf{E}.VT$ *and the additional tuple* $e_5 = (Dave, SWE, 5, [04/11, 09/26))$.

The result of query $\pi_{PID,LR,\mathbf{P}.VT-\mathbf{E}.VT/Diff}(\{p_1\} \bowtie_{LR=Level} \{e_1\}) \bowtie_{LR \leq Level \wedge Diff\ overlaps\ \mathbf{E}.VT} (\{e_5\})$ *depends on the representation of the difference result since the truth value of predicate* overlaps *depends on the representation of the input time intervals.*

At reference time $02/04$, *the expected result interval of difference* $[02/04, 07/20) - [03/08, now)$ *is time interval* $[02/04, 07/20)$ *(cf. Example 2). Then, predicate* ($[02/04, 07/20)$ overlaps $[04/11, 09/26)$) *is* true *and we get result tuple*

$$v_{exp} = (500, 4, [02/04, 07/20), Dave, SWE, 5, [04/11, 09/26))$$

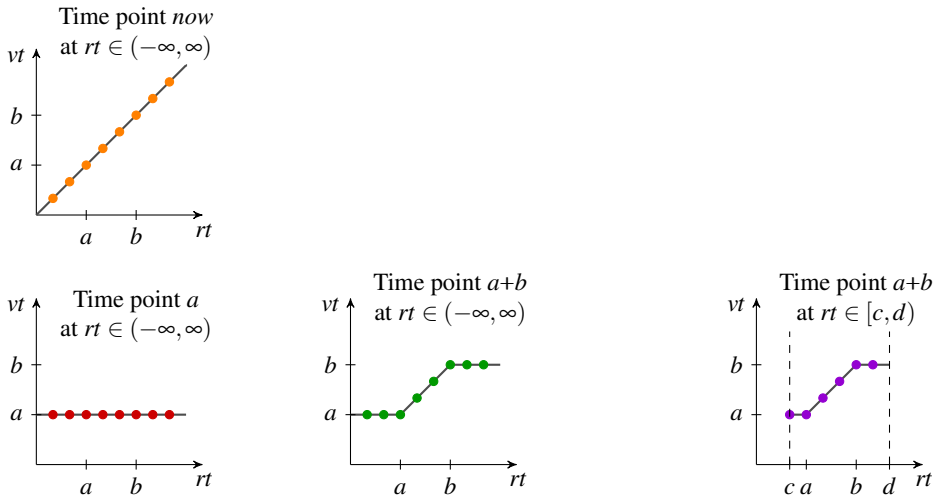
At reference time $02/04$, *an unexpected result would be the two time intervals* $[02/04, 03/08)$ *and* $[03/08, 07/20)$, *which splits the expected time interval into sub-time intervals. For result interval* $[02/04, 03/08)$, *predicate* ($[02/04, 03/08)$ overlaps $[04/11, 09/26)$) *is* false *and for result interval* $[03/08, 07/20)$, *predicate* ($[03/08, 07/20)$ overlaps $[04/11, 09/26)$) *is* true. *We get*

result tuple

$$v_{unexpected} = (500, 4, [03/08, 07/20), Dave, SWE, 5, [04/11, 09/26))$$

Clearly, result tuple $v_{\text{unexpected}}$ *is incorrect at reference time* $02/04$ *since it states that employee Ann cannot work on the project in* $[03/08, 07/20)$ *(meaning of the difference result) when, in fact, employee Ann cannot work on the project for its whole timeframe* $[02/04, 07/20)$*. As a consequence, the query tries to find employees for the incorrect not-covered project timeframe with the join.*

Figure 1.5 gives an overview of the time domains for ongoing time points. Time domain $\mathscr{T} \cup \{now\}$ [CDI+97] consists of fixed time points a and the ongoing time point *now* (Figure 1.5a). Ongoing time domain Ω [MB20b] generalizes time domain $\mathscr{T} \cup \{now\}$ and consists of ongoing time points a+b (Figure 1.5b). The meaning of ongoing time point a+b is *not earlier than a, but not later than b*. It is equal to time point a up to reference time a, it is equal to the reference time between reference times a and b, and it is equal to time point b from reference time b on. These time domains are not sufficient to represent the function results, such that they consist of the expected time intervals at every reference time. The reason is that ongoing time points (the start and end points of ongoing time intervals) instantiate to a time point at every possible reference time rt, i.e., $rt \in (-\infty, \infty)$, as illustrated in Figure 1.5.



(a) Domain $\mathscr{T} \cup \{now\}$. (b) Domain Ω. (c) Ongoing time points at some rt only.

Figure 1.5: Overview of ongoing time points.

Instead of ongoing time points that can be instantiated *at all possible* reference times, we need ongoing time points that can be instantiated *at some* reference times only (cf. Figure 1.5c). We do this by representing the function result as a combination of ongoing intervals and the reference time when the ongoing interval is part of the result. For instance, we represent the result of the difference $[02/04, 07/20) - [03/08, now)$ as follows:

VT	RT
$[02/04, 07/20)$	$\{(-\infty, 03/09)\}$
$[02/04, 03/08)$	$\{[03/09, \infty)\}$
$[now, 07/20)$	$\{[03/09, \infty)\}$

The first entry means that ongoing interval $[02/04, 07/20)$ is part of the difference result exactly at the reference times in $\{(-\infty, 03/09)\}$ (cf. ongoing interval T_1' in Figure 1.4c).

We can freely restrict the reference time *RT* of an ongoing result interval since the reference time is an auxiliary time dimension that is only used to correctly interpret ongoing values, such as time points, time intervals, and tuples. The reference time cannot be used in functions and predicates except when combining an *RT* value with other *RT* values or with the results of predicates via logical connectives, i.e., conjunction, disjunction, and negation (cf. relational algebra operators in Section 1.2). For the combination, only the reference time points in the *RT* value are of interest, but not its representation.

Representing function results as combinations of ongoing intervals and the reference time when the ongoing interval is part of the result allows us to keep the simplicity and intuitive meaning that ongoing time points and time intervals of existing time domains have while being able to correctly represent the function result for all reference times.

We leverage ongoing relations (cf. first challenge) to store the function results in relations: the ongoing time interval is stored in its own attribute and the tuple's reference time *RT* is restricted to the reference time when the ongoing interval is part of the result.

Relational Algebra Operators. Relational algebra operators need to correctly restrict the reference time *RT* of the result tuples in order to retain their semantics at every reference time.

As discussed before in Section 1.2, this thesis proposes *ongoing relations* that associate each tuple with a reference time attribute *RT*. The *RT* value includes the reference times when *now* can be instantiated in the tuple and the tuple belongs to the instantiated relations. Ongoing relations are the input and the result of relational algebra operators.

Relational algebra operators have to take the following three requirements into account when determining their result:

(1) Tuples belong to the input relation at some reference times only (RT value).

(2) Predicates select tuples at some reference times only.

(3) Function results consist of ongoing values that are part of the result at some reference times only.

Relational algebra operators can satisfy these requirements by restricting the reference time RT of the result tuples accordingly. As an example, consider the inner join operator $\mathbf{R} \bowtie_\theta \mathbf{S}$. The semantics of an inner join are that a tuple $r \in \mathbf{R}$ is joined with a tuple $s \in \mathbf{S}$ if the join predicate $\theta(r,s)$ applied to tuples r and s is satisfied. For ongoing relations $\mathbf{R}$ and $\mathbf{S}$, this means that a joined tuple $(r \circ s)$ belongs to the join result *at the reference times* when input tuples r and s belong to the input relations ($r.RT$ and $s.RT$) and the join predicate $\theta(r,s)$ is *true*. It is not a binary decision anymore if a joined tuple belongs to the result; instead the join operator restricts the RT value of the joined tuple $(r \circ s)$ to $RT = r.RT \wedge s.RT \wedge \theta(r,s)$, consisting of the reference times when the joined tuple belongs to the result.

1.3 Contributions

This thesis makes three main contributions to the database field:

- It introduces a framework that evaluates predicates, functions, and relational algebra operators to results that remain valid as time passes by. As query results, the thesis proposes *ongoing relations* that associate each tuple with a reference time attribute RT. The RT attribute includes the reference times when *now* can be instantiated in the tuple and the tuple belongs to the relation. The reference time attribute accommodates the selection of tuples at some reference times only.

- It defines functions on ongoing time intervals, such that their results consist of the expected time intervals at every reference time. As function results, the thesis proposes pairs consisting of ongoing intervals and the reference time when the ongoing interval is part

of the result. To store these results in relations, we leverage ongoing relations with a single reference time attribute that integrates the restrictions from the results of all interval functions.

- It proposes an aggregation operator for ongoing relations that correctly and efficiently groups and aggregates the tuples in an ongoing relation at every reference time. The aggregation operator adjusts the input tuples, such that the tuples that are part of the same group have the same grouping attribute values and the same reference time RT. The aggregation result is an ongoing relation that remains valid as time passes by.

The research methodology that has been adopted for each part of this thesis starts with a problem given from real world followed by an analysis and precise definition of the problem. The solution to a problem and its properties are studied, elaborated analytically, and then implemented. Large parts of this thesis have been implemented into the open source database system PostgreSQL and made available as open source (https://www.ifi.uzh.ch/dbtg/research.html). The implementation is extensively evaluated and compared with the state-of-the-art approaches to confirm the analytical results of the solution. The rest of this section elaborates the contributions of this thesis in more detail with examples.

1.3.1 Framework for Operations to Get Results that Remain Valid

The first contribution of this thesis is a framework that evaluates predicates, functions, and relational algebra operators on ongoing data types to results that remain valid as time passes by. The key idea is to evaluate the operations at every possible reference time and represent the results as ongoing data types. Ongoing values are kept uninstantiated during query processing and in the result.

In this framework, operations on ongoing data types are defined as follows. Given an operation op on ongoing data types with input arguments $i_1, \ldots, i_n$ and the corresponding operation op^F on fixed data types, operation op evaluates to a result that, at each reference time rt, is equal to the result obtained by the corresponding operation op^F on fixed data types:

$$\forall rt(\|op(i_1, \ldots, i_n)\|_{rt} = op^F(\|i_1\|_{rt}, \ldots, \|i_n\|_{rt}))$$

The bind operator $\|\cdot\|_{rt}$ replaces all occurrences of *now* with the reference time rt. We use the ^{F}-superscript for operations on fixed data types.

Example 4. *Consider predicate* overlaps *for ongoing time intervals and the corresponding predicate* overlaps^F *for fixed time intervals. Predicate* overlaps^F *for fixed time intervals is defined as usual:* $[a,b)$ overlaps^F $[c,d) = a <^F d \wedge^F c <^F b$. *Then, predicate* overlaps *for ongoing time intervals* T_1 *and* T_2 *is defined as:* $\forall rt(\|T_1 \text{ overlaps } T_2\|_{rt} = \|T_1\|_{rt} \text{ overlaps}^F \|T_2\|_{rt})$.

Predicate $([05/14, now)$ overlaps $[07/20, 10/29))$ *is* true *at the reference times in* $\{[07/21,\infty)\}$ *and* false *otherwise:* $(\text{true} \Leftrightarrow rt \in \{[07/21,\infty)\})$. *At each reference time, its truth value is equal to the truth value obtained by evaluating* overlaps^F *on the instantiated input arguments, i.e., at every reference time* rt, $(\text{true} \Leftrightarrow rt \in \{[07/21,\infty)\})$ *is equal to* $\|[05/14, now)\|_{rt}$ overlaps^F $\|[07/20, 10/29)\|_{rt}$. *Figure 1.6 illustrates the equality for reference times* 07/01 *and* 08/01.

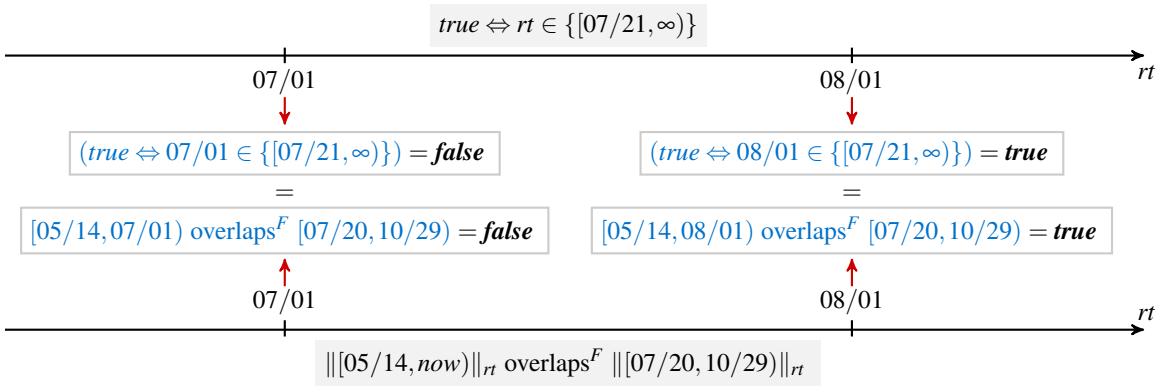


Figure 1.6: The result of predicate *overlaps* remains valid.

Our approach supports the commonly used predicates on time points and time intervals also for ongoing time points and ongoing time intervals. For the logical connectives and predicates offered by PostgreSQL, we provide implementations as their ongoing equivalent. This includes the logical connectives and predicates in Table 1.1.

Table 1.1: Commonly used predicates and logical connectives supported by our approach for ongoing data types.

Logical connectives	$\wedge$	$\vee$	$\neg$	
Predicates for ongoing time points	$<$	$\leq$	$>$	$\geq$
	$=$	$\neq$		
Predicates for ongoing time intervals	before	after	starts	finishes
	$=$	$\neq$	overlaps	during
	adjacent	overleft	overright	

We propose time domain $\Omega = \{a{+}b | a, b \text{ as fixed time points } \wedge a \leq b\}$ as the time domain for ongoing time points. Ongoing time point $a{+}b$ is illustrated in Figure 1.7. Its intuitive meaning is *not earlier than a, but not later than b*. For instance, $03/08{+}07/20$ means *not earlier than* $03/08$ *but not later than* $07/20$.

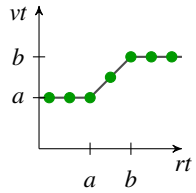


Figure 1.7: Illustration of ongoing time point $a{+}b$.

A fixed time point a, current time point *now*, a growing time point $+a$ and a limited time point $+b$ can all be expressed as ongoing time points of the form $a{+}b$. This is illustrated in Figure 1.8.

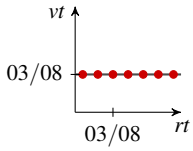
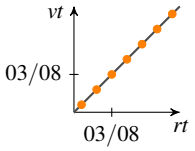
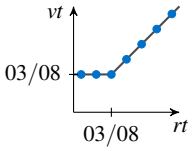
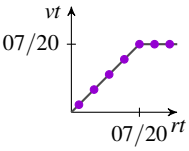
	Fixed time point	**Time point *now***	**Growing time point**	**Limited time point**
Notation				
- as $a{+}b$	$a{+}a$	$-\infty{+}\infty$	$a{+}\infty$	$-\infty{+}b$
- short	a	*now*	$a+$	$+b$
Meaning	time point a	the *current* time point	not earlier than a, possibly later	possibly earlier, but not later than b
Example	$03/08$	*now*	$03/08+$	$+07/20$
Semantics				

Figure 1.8: Ongoing time points expressed as $a{+}b$.

The benefit of time domain Ω is that it is closed for min and max, i.e., the evaluation of min and max on Ω again yields time points of Ω. Functions min and max are useful auxiliary functions to efficiently implement functions like the intersection, the difference, and the union function for ongoing time intervals and predicates like *overlaps*. For instance, predicate $[t_s, t_e)$ *overlaps* $[\tilde{t}_s, \tilde{t}_e)$ can then be implemented as $(\max(t_s, \tilde{t}_s) < \min(t_e, \tilde{t}_e))$ instead of using the usual definition of the predicate. In the ongoing case, the former is more efficient since min and max can efficiently determine the values of a and b of the result ongoing time point $a{+}b$ and this is faster than evaluating several predicates and conjunctions on ongoing values.

To represent the results of relational algebra operators, this thesis proposes *ongoing relations*. Each tuple in an ongoing relation is associated with a reference time attribute RT that contains

the reference times when *now* can be instantiated in the tuple and the tuple belongs to the instantiated relations. The value of the RT attribute is set by the database system and restricted by predicates on ongoing attributes. The RT value is a set of reference time points that we represent as set of time intervals. How the reference time points are grouped into time intervals is not relevant for the semantics of the RT attribute since it is an internal attribute that cannot be used in functions and predicates except when combining an RT value with other RT values or with the results of predicates via logical connectives, i.e., conjunction, disjunction, and negation. For the combination, only the reference time points in the RT value are of interest, but not its representation.

Example 5. *The result of query Q_1 in Example 1 is the ongoing relation $\mathbf{V_1}$ shown in Figure 1.9. Each tuple is associated with a* reference time attribute *RT that consists of the reference times when the tuple belongs to the relation. The intersection result is discussed in Example 6.*

$\mathbf{V_1}$

	PID	Name	Role	$\mathbf{P}.VT \cap \mathbf{E}.VT$	RT
v_{11}	500	Ann	SWE	$[03/08, +07/20)$	$\{[03/09, \infty)\}$
v_{12}	500	John	SRE	$[03/08, 07/20)$	$\{(-\infty, \infty)\}$
v_{13}	501	Bob	SWE	$[07/20, now)$	$\{[07/21, \infty)\}$
v_{14}	501	Eve	SRE	$[07/20, +10/29)$	$\{[07/21, \infty)\}$

Figure 1.9: The result of query Q_1 is an ongoing relation that remains valid as time passes by.

The RT value of each tuple in ongoing relation $\mathbf{V_1}$ consists of the reference times when the join predicate $\theta = (LR = Level \wedge \mathbf{P}.VT$ overlaps *$\mathbf{E}.VT)$ is* true*. At these reference times, the tuple belongs to the result relation. For instance, $\theta(p_2, e_3) = (6 = 6 \wedge [05/14, now)$* overlaps *$[07/20, 10/29))$ is* true *from reference time $07/21$ on (cf. predicates on ongoing values in Section 1.2). Exactly these reference times are included in the RT value of result tuple v_{14}: $v_{14}.RT = \{[07/21, \infty)\}$.*

For the newly introduced ongoing relations, we provide a relational algebra. The input and result of each operator are ongoing relations that preserve ongoing time points and that remain valid as time passes by. The relational algebra operators restrict the reference time of the result tuples based on the reference time of the input tuple(s) and the semantics of the relational algebra operator. As an example, the inner join $\mathbf{R} \bowtie_\theta \mathbf{S}$ is equivalent to the following ongoing relation:

$$\mathbf{R} \bowtie_\theta \mathbf{S} \equiv \{(r.\mathbf{A}, s.\mathbf{B}, RT) | r \in \mathbf{R} \wedge s \in \mathbf{S} \wedge RT = (r.RT \wedge s.RT \wedge \theta(r,s))\}$$

This means that for each tuple $r \in \mathbf{R}$ and $s \in \mathbf{S}$, the ongoing relation contains a joined tuple $(r \circ s)$ whose reference time RT consists of the reference times when both tuple r and tuple s belong to

the respective input relation (r.RT and s.RT) and the reference times when join predicate $\theta(r,s)$ is *true* for tuples r and s. This makes sense since tuples can only be joined if both tuples belong to their respective input relation and it is consistent with the semantics of the inner join since only tuples that satisfy the join predicate belong to the result relation.

Example 6. *The result ongoing relation* $\mathbf{V_1}$ *in Figure 1.9 preserves ongoing time points and remains valid as time passes by.*

The intersection $\mathbf{P}.VT \cap \mathbf{E}.VT$ *states the timeframe when an employee can work on a project. The result of the intersection is an ongoing time interval whose start and end points are ongoing time points of our proposed time domain* Ω*. For instance, the assignment timeframe in tuple* v_{11} *is ongoing time interval* $p_1.VT \cap e_1.VT = [03/08, +07/20)$*. Its intuitive meaning is* from $03/08$ until possibly earlier, but not later than $07/20$*. At reference time* $04/01$*, tuple* v_{11} *means that Ann can work as SWE on the project with id 500 from* $03/08$ *until* $04/01$ *and at reference time* $08/01$*, it means that Ann can work as SWE on the project with id 500 from* $03/08$ *until* $07/20$*.*

At each reference time, ongoing relation $\mathbf{V_1}$ *is equal to the result obtained when evaluating query* Q_1 *on the instantiated input relations. Figure 1.10 illustrates this for the reference times* $04/01$ *and* $08/01$*. The expected query results at these reference times have been discussed in Example 1 and given in Figure 1.3. At reference time* $04/01$*, ongoing relation* $\mathbf{V_1}$ *consists of the tuples* v_{11} *and* v_{12} *since reference time* $04/01$ *is contained in their RT values. This is consistent with the expected query result. The ongoing intersection intervals instantiate to the time intervals in the expected query result. At reference time* $08/01$*, ongoing relation* $\mathbf{V_1}$ *consists of the tuples* v_{11}*,* v_{12}*,* v_{13}*, and* v_{14} *since reference time* $08/01$ *is contained in their RT values. This is consistent with the expected query result. The ongoing intersection intervals instantiate to the time intervals in the expected query result.*

1.3.2 Interval Functions with Expected Result Intervals

The second contribution are functions on ongoing intervals that evaluate to the expected result intervals at every reference time. The expected time intervals at a reference time are the time intervals one would expect if all occurrences of *now* had been replaced by the reference time. As an example, for the difference function $T_1 - T_2$ the expected time intervals are the maximal sub-intervals of T_1 that do not overlap with T_2. We provide solutions for the intersection, the difference, and the union function since these are the standard functions and building blocks for processing time intervals. We refer to these functions as *interval functions*.

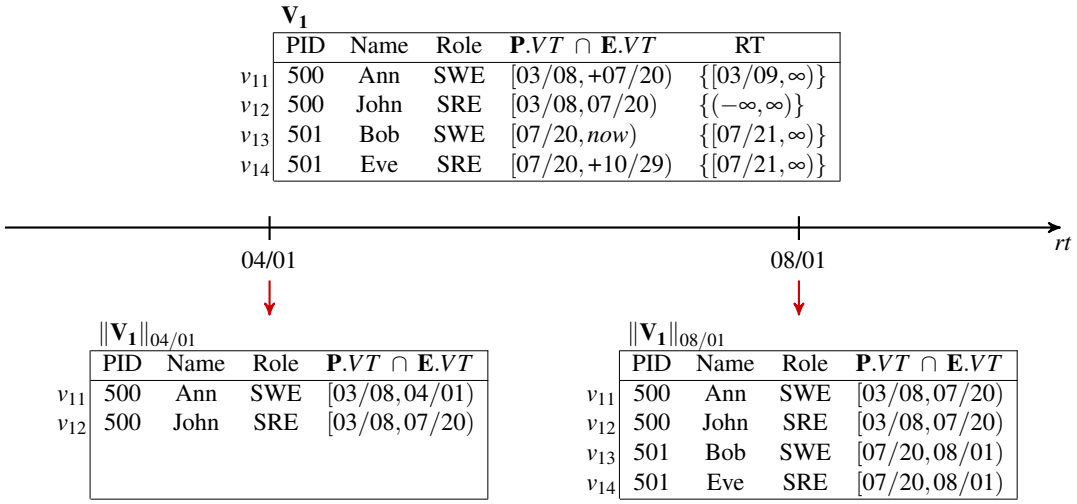


Figure 1.10: Ongoing relation $\mathbf{V_1}$ represents the correct query result at every reference time.

We apply the framework introduced in Section 1.3.1 to the interval functions to get function results that remain valid as time passes by. We define an interval function based on the corresponding function for fixed time intervals. For instance, at each reference time rt, the difference function on ongoing time intervals, $T_1 - T_2$, evaluates to the same result to which the difference function $-^F$ for fixed time intervals evaluates: $\forall rt(\|T_1 - T_2\|_{rt} = \|T_1\|_{rt} -^F \|T_2\|_{rt})$.

We represent the function results as a combination of ongoing intervals and the reference time RT when the ongoing interval is part of the result. This representation gives us the flexibility needed to represent the expected time intervals at every reference time. Interval functions return sets of (ongoing interval, RT)-pairs as illustrated in Figure 1.4 and Figure 1.11.

Our implementation uses ongoing relations to store the results of interval functions in databases. An ongoing relation associates each tuple with a reference time attribute RT whose value are the reference times when the tuple belongs to the relation (cf. Section 1.3.1). We store each (ongoing interval, RT)-pair in its own tuple. The ongoing time interval is stored in an attribute corresponding to the function; the reference time of the tuple is restricted with the pair's reference time. This ensures that the tuple belongs to the relations exactly at the reference times when the ongoing interval is part of the function result.

Example 7. *Consider query Q_2 in Example 1 with sub-query Q_{2a} that retrieves for each project with at least one eligible employee the employees that have exactly the required employment level of the project. For each potential project assignment, the timeframe* $\mathbf{P}.VT - \mathbf{E}.VT$ *when*

the employee cannot work on the project is of interest:

$$Q_{2a} = D \leftarrow \pi_{PID,LR,Name,(\mathbf{P}.VT\ -\ \mathbf{E}.VT)/Diff}(\mathbf{P} \bowtie_{LR=Level} \mathbf{E})$$

We write $\pi_{X/C}$ to rename X to C. We added the name of the employee to the projection list to more easily distinguish between the tuples in our illustration.

Figure 1.11 illustrates the result of the difference function and its storage in the result ongoing relation. The hatched area refers to the part of $\mathbf{P}.VT$ that intersects with $\mathbf{E}.VT$ and thus, is not part of the difference result. The difference function is evaluated on each input tuple. As an example, the result of the difference $\mathbf{P}.VT - \mathbf{E}.VT$ for input tuple j_1 consists of three (ongoing interval, RT)-pairs. For each of the result pairs, a result tuple is produced. The difference result for input tuple j_1 is stored in the tuples d_{11}, d_{12}, and d_{13}. The ongoing interval is stored in the attribute Diff *and the reference time RT of the input tuple is restricted with the RT value of the result pair. The RT value of result tuple d_{11} is the conjunction of $j_1.RT$ and the result pair's $RT = \{(-\infty, 03/09)\}$:*

$$\begin{aligned} d_{11}.RT &= j_1.RT \wedge \{(-\infty, 03/09)\} \\ &= \{(-\infty, \infty)\} \wedge \{(-\infty, 03/09)\} \\ &= \{(-\infty, 03/09)\} \end{aligned}$$

Our approach flattens relational algebra operators with multiple and nested functions into nested projections, so that each projection list includes one interval function at most. This is an effective mechanism to handle several functions with each returning *a set of result pairs* and still being able to express the query with standard SQL. For multiple functions the flattening is equivalent to the cross product of the result intervals. For nested functions it is equivalent to applying the enclosing function to each result interval of the nested function.

Example 8. *Query Q_3 retrieves for each project with at least one eligible employee the employees that have exactly the required employment level of the project. For each potential project assignment, the timeframe $\mathbf{P}.VT \cap \mathbf{E}.VT$ when the employee can work on the project and the timeframe $\mathbf{P}.VT - \mathbf{E}.VT$ when the employee cannot work on the project are of interest.*

$$Q_3 = \pi_{PID,Name,Role,\mathbf{P}.VT \cap \mathbf{E}.VT/interVT,\mathbf{P}.VT\ -\ \mathbf{E}.VT/diffVT}(\mathbf{P} \bowtie_{LR=Level} \mathbf{E})$$

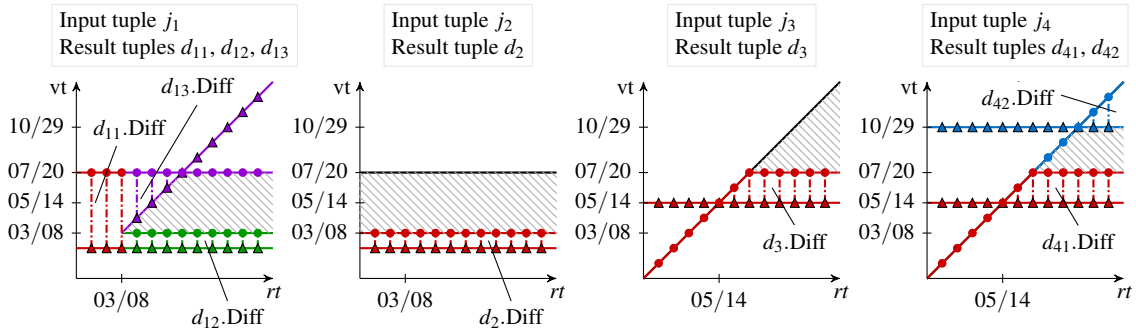
Joined relation $J \rightarrow \mathbf{P} \bowtie_{LR=\text{Level}} \mathbf{E}$ as input

$J \rightarrow \mathbf{P} \bowtie_{LR=\text{Level}} \mathbf{E}$

	PID	LR	$\mathbf{P}.VT$	Name	Role	Level	$\mathbf{E}.VT$	RT
j_1	500	4	$[02/04, 07/20)$	Ann	SWE	4	$[03/08, now)$	$\{(-\infty, \infty)\}$
j_2	500	4	$[02/04, 07/20)$	John	SRE	4	$[03/08, 08/23)$	$\{(-\infty, \infty)\}$
j_3	501	6	$[05/14, now)$	Bob	SWE	6	$[07/20, now)$	$\{(-\infty, \infty)\}$
j_4	501	6	$[05/14, now)$	Eve	SRE	6	$[07/20, 10/29)$	$\{(-\infty, \infty)\}$

Projection with difference function $\pi_{\text{PID},LR,Name,(\mathbf{P}.VT\ -\ \mathbf{E}.VT)/\text{Diff}}(\mathbf{J})$

(a) Calculation of the difference $\mathbf{P}.VT\ -\ \mathbf{E}.VT$



(b) Storing the difference result in the ongoing relation

$D \leftarrow \pi_{\text{PID,LR,Name},(\mathbf{P}.VT\ -\ \mathbf{E}.VT)/\text{Diff}}(\mathbf{J})$

	PID	LR	Name	Diff	RT
d_{11}	500	4	Ann	$[02/04, 07/20)$	$\{(-\infty, 03/09)\}$
d_{12}	500	4	Ann	$[02/04, 03/08)$	$\{[03/09, \infty)\}$
d_{13}	500	4	Ann	$[now, 07/20)$	$\{[03/09, \infty)\}$
d_2	500	4	John	$[02/04, 03/08)$	$\{(-\infty, \infty)\}$
d_3	501	6	Bob	$[05/14, +07/20)$	$\{(-\infty, \infty)\}$
d_{41}	501	6	Eve	$[05/14, +07/20)$	$\{(-\infty, \infty)\}$
d_{42}	501	6	Eve	$[10/29, now)$	$\{(-\infty, \infty)\}$

Figure 1.11: The difference function returns (ongoing interval, RT)-pairs and the difference result is stored in an ongoing relation with a tuple for each pair.

The projection in query Q_3 contains the two interval functions intersection *and* difference. *We flatten the projection into nested projections with a projection with the intersection function and a projection with the difference function:*

$$Q_3^{flatten} = \pi_{PID,Name,Role,interVT,diffVT}(\pi_{*,interVT,\mathbf{P}.VT\ -\ \mathbf{E}.VT/diffVT}(\pi_{*,\mathbf{P}.VT\ \cap\ \mathbf{E}.VT/interVT}(\mathbf{P} \bowtie_{LR=Level} \mathbf{E})))$$

$Q_3^{flatten}$ can then be expressed with standard SQL. For the difference and intersection function, we use our implementation as set-returning functions, which return one record per result tuple [Pos20c].

```
SELECT PID, Name, Role, interVT, diffVT, (RT ∧ diffRT) as RT
FROM (SELECT PID, LR, PVT, Name, Role, Level, EVT,
             interVT, (diff(PVT,EVT)).*,(RT ∧ interRT) as RT
      FROM (SELECT *, (intersect(PVT, EVT)).*
            FROM (SELECT PID, LR, P.VT as PVT, Name, Role, Level,
                         E.VT as EVT, (P.RT ∧ E.RT) as RT
                  FROM P JOIN E ON LR = Level
                 ) as innerJoin
           ) as intersectFctResult
     ) as diffFctResult;
```

The inner-most select query calculates the inner join. Predicates on fixed attributes retain their standard behavior and are used as usual (LR = Level). The enclosing select query uses the set-returning intersection function to calculate the intersection of the valid times. The returned result record of the function is of the form (interVT, interRT) *and is decomposed with the .∗ syntax into its two attributes. The enclosing select query (1) stores the intersection result interval in the* interVT *attribute and sets the reference time of the result tuple to the conjunction of* interRT *and the input tuple's RT and (2) calculates the difference of the ongoing valid times with the set-returning difference function. The outer most select query stores the difference result interval in the* diffVT *attribute and restricts the reference time of the input tuple with* diffRT.

1.3.3 Aggregation on Ongoing Relations

The aggregation ${}_{\mathbf{G}}\vartheta_{\phi}(\mathbf{R})$ divides the tuples in relation **R** into groups with equal values of the grouping attributes **G** and aggregates each group with aggregate functions ϕ into a single tuple.

The third contribution is the aggregation operator on ongoing relations that correctly and efficiently groups and aggregates the tuples in an ongoing relation. The result is an ongoing relation that contains for each group a single, aggregated tuple. Aggregating data is an important and often performed task in database systems to summarize data and retrieve interesting, statistical information about the data.

Our focus is to determine the correct groups for a tuple depending on the reference time. Both fixed attributes without ongoing values and ongoing attributes with ongoing values can be used as grouping attributes. The key challenge is to determine the aggregation groups: (1) since ongoing values change as time passes by, tuples can be equal at some reference times only, and (2) tuples with equal grouping attribute values might belong to the relation at different reference times. The first case occurs when the grouping attributes contain ongoing attributes; the second case occurs for ongoing relations due to the presence of the *RT* attribute. As a consequence, a tuple might be part of a group at some reference times only and it might be part of different groups at disjoint reference times.

We consider aggregate functions on fixed attributes only; aggregate functions on ongoing attributes are beyond the scope of this thesis and part of future research. All aggregate functions that are supported by current database systems can also be used with our solution. Examples are *count*, *min*, *max*, *sum*, and *avg*.

We apply the framework introduced in Section 1.3.1 to the aggregation operator to get results that remain valid as time passes by. At each reference time *rt*, the aggregation ${}_{\mathbf{G}}\vartheta_{\phi}(\mathbf{R})$ on ongoing relation $\mathbf{R}$ with grouping attributes $\mathbf{G}$ and aggregate functions ϕ evaluates to a relation that is equal to the result obtained by evaluating the aggregation operator for fixed relations on the instantiated input relation: $\forall rt(\|{}_{\mathbf{G}}\vartheta_{\phi}(\mathbf{R})\|_{rt} = {}_{\mathbf{G}}\vartheta^{F}_{\phi}(\|\mathbf{R}\|_{rt}))$.

The thesis proposes an aggregation algorithm for ongoing relations that we integrated into the query processing pipeline of the PostgreSQL database system. Conceptually, the algorithm first divides the input tuples into fixed groups, each containing the tuples with equal fixed grouping attribute values, then further divides each fixed group into ongoing groups according to the ongoing grouping attributes and the reference time attribute *RT*, and finally aggregates each ongoing group into a single result tuple with the aggregate functions.

The aggregation algorithm incrementally calculates the fixed groups, the ongoing groups of a fixed group, and the aggregate values of an ongoing group and intertwines their calculation. The algorithm first sorts the input tuples according to the fixed grouping attributes to consecutively process tuples that are part of the same fixed group. Within a fixed group, the ongoing groups are incrementally built by manipulating the group's reference time *RT*. An ongoing group is represented with (1) a single master tuple that provides the fixed and ongoing attribute values and the reference time *RT* of the group and (2) the intermediate aggregate values. Each subsequent tuple t splits each ongoing group g into two ongoing groups: the first ongoing group g_1 that consists of the reference times when the group and the tuple are equal according to the ongoing

grouping attributes $\mathbf{G}_o$ and have common reference times RT, $g.\mathbf{G}_o = t.\mathbf{G}_o \wedge g.RT \wedge t.RT$, and the second ongoing group g_2 whose reference time is the reference time of the original ongoing group minus the reference time of g_1. A new ongoing group g_3 with tuple t as its master tuple is created for the reference times when the tuple is not equal to any ongoing group. These reference times are included in the group's RT value. Whenever a tuple is conceptually added to an ongoing group (groups g_1 and g_3), the aggregate values of the group are updated. Once all tuples of a fixed group have been processed, the aggregate values of the ongoing groups are finalized and a result tuple for each ongoing group is produced.

The aggregation algorithm leverages the existing, optimized strategies of the database system to determine groups and aggregate values. The algorithm uses the existing grouping mechanisms to determine tuples with equal fixed grouping attribute values and it uses the existing aggregate calculation strategies to incrementally determine the aggregate values of each ongoing group.

Calculating the aggregation groups and the aggregate values incrementally avoids materializing the tuples that belong to an aggregation group and keeps the memory consumption per ongoing group constant, independent of the size of the group. This is especially important for the aggregation on ongoing relations since a tuple might belong to several ongoing groups within a fixed group.

1.4 Organization of the Thesis

This thesis is based on a collection of papers. A bibliography for all chapters is given at the end of the thesis.

Chapter 2 Query Results over Ongoing Databases Whose Results Remain Valid as Time Passes By

> Yvonne Mülle and Michael H. Böhlen, "Query Results over Ongoing Databases Whose Results Remain Valid as Time Passes By (Extended Version)", *Technical Report CoRR*, https://arxiv.org/pdf/2001.05722.pdf, 2020
>
> based on the paper: Yvonne Mülle and Michael H. Böhlen, "Query Results over Ongoing Databases Whose Results Remain Valid as Time Passes By", in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1429 - 1440, 2020.

Chapter 3 Functions on Ongoing Intervals

> Yvonne Mülle and Michael H. Böhlen, "Functions on Ongoing Intervals", [ready for submission]

Chapter 4 Aggregation on Ongoing Relations

> Yvonne Mülle and Michael H. Böhlen, "Aggregation on Ongoing Relations", [ready for submission]

CHAPTER 2

Query Results over Ongoing Databases that Remain Valid as Time Passes By

Abstract

Ongoing time point *now* is used to state that a tuple is valid from the start point onward. For database systems ongoing time points have far-reaching implications since they change continuously as time passes by. State-of-the-art approaches deal with ongoing time points by instantiating them to the reference time. The instantiation yields query results that are only valid at the chosen time and get invalidated as time passes by.

We propose a solution that keeps ongoing time points *uninstantiated* during query processing. We do so by evaluating predicates and functions at all possible reference times. This renders query results independent of a specific reference time and yields results that remain valid as time passes by. As query results, we propose *ongoing relations* that include a *reference time attribute*. The value of the reference time attribute is restricted by predicates and functions on ongoing attributes. We describe and evaluate an efficient implementation of ongoing data types and operations in PostgreSQL.

2.1 Introduction

Data that are associated with a valid time interval [JS09] are present in real-world applications that deal with employment contracts, insurance policies, software bugs, etc. The ongoing time point *now* is commonly used to state that the contract, policy, bug, etc. is valid from the start point onward.

The ongoing time point *now* changes its value when time passes by and the reference time is used to determine the value. At each reference time, *now* instantiates to the time point equal to the reference time. For example, at reference time $08/15$, *now* instantiates to time point $08/15$ and at reference time $08/16$, it instantiates to time point $08/16$. Throughout the paper, we use time points in the mm/dd format relative to 2019: time point $08/15$ denotes August 15, 2019.

A key assumption of database systems is that query results only get outdated if data is modified explicitly. This happens if data is inserted, updated, or deleted. The assumption no longer holds if *now* is stored in the database or when queries are evaluated on databases with ongoing time points [CDI+97, TJS04, ASTS13]. In this case, query results get also outdated as a result of time passing by. This has significant drawbacks. First, query results, including materialized views and cached query results, must be re-computed before they can be accessed. Second, because ongoing time points are replaced by fixed time points, it is impossible for applications to identify result time points that change when time passes by.

This paper proposes an elegant and efficient solution that preserves ongoing time points in query results and that evaluates queries at all possible reference times to get results that remain valid as time passes by. Formally, given a database D with ongoing time points and a query Q, we want to compute a query result $Q(D)$, such that at every possible reference time rt, the query result is equivalent to the result obtained by instantiating *now* in D and evaluating the query on the instantiated database: $\forall rt(\|Q(D)\|_{rt} \equiv Q(\|D\|_{rt}))$. The bind operator $\|\cdot\|_{rt}$ replaces all occurrences of *now* with the reference time rt.

To support queries with predicates and functions on ongoing attributes, the key challenges are (1) the evaluation of queries to results that remain valid as time passes by and (2) the representation of these results.

To get results that remain valid, we keep ongoing time points uninstantiated. We define six core operations predicate $<$, functions min and max, and the logical connectives $\wedge$, $\vee$, $\neg$. At each reference time, their results are equal to the results obtained by the corresponding operations for

fixed data types on the instantiated input arguments. We provide equivalences for the core operations and for additional operations that are expressed with the core operations. The equivalences are used for an efficient implementation. We represent the results of predicates and logical connectives as *ongoing booleans*, i.e., booleans whose truth value depends on the time. The results of relational algebra operators are represented as *ongoing relations* that include a reference time attribute RT. The value of RT includes the reference times when a tuple belongs to the instantiated relations. The reference time of a tuple is restricted by predicates in queries. We represent the value of the RT attribute with a finite set of fixed time intervals. Thus, only predicates that evaluate to booleans that change their value a finite number of times are allowed. The tuples in base ongoing relations have a trivial reference time, i.e., $RT = \{(-\infty, \infty)\}$. Tuples with an empty reference time, i.e., $RT = \{\}$, are deleted.

Our technical contributions are the following:

- We propose the ongoing time domain Ω for *ongoing time points*. The time domain is closed for min and max, i.e., the evaluation of min and max on Ω again yields an ongoing time point of Ω.
- We define predicates, functions and logical connectives that keep ongoing time points uninstantiated during query processing.
- We introduce *ongoing relations* with a reference time attribute to represent query results that remain valid as time passes by. The value of the RT attribute is set by the database system and restricted by predicates on ongoing attributes.
- We define the relational algebra for ongoing relations. The result of each operator is an ongoing relation that remains valid as time passes by.
- We describe an efficient implementation of ongoing data types and operations on these data types in the kernel of PostgreSQL.

The paper is organized as follows. Section 2.2 introduces our running example. Section 2.3 discusses related work. Section 2.4 provides preliminaries. We define the time domain for ongoing time points in Section 2.5. Predicates and functions on ongoing time points and time intervals whose results remain valid are discussed in Section 2.6. Section 2.7 introduces ongoing relations and defines a relational algebra on them. Section 2.8 discusses the implementation of our solution in PostgreSQL. The evaluation is described in Section 2.9. Section 2.10 concludes the paper and points to future research.

2.2 Running Example

Consider a company that keeps track of bugs associated with the individual components of its email service. Prioritized bugs have fixed start points that indicate when the bug was discovered and fixed end points that indicate the deadline for resolving the bug internally. Deprioritized bugs have fixed start points but end points that keep increasing. These end points are *ongoing*. A bug is open iff it has been discovered but not yet resolved internally. Once a bug has been resolved internally, its fix will be deployed in a future patch to the production servers. The patches for the components of the email service are pre-scheduled. Selected relations of our running example are shown in Figure 2.1 and discussed below.

B

	BID	C	VT	RT
b_1	500	Spam filter	$[01/25, now)$	$\{(-\infty, \infty)\}$
b_2	501	Spam filter	$[03/30, 08/21)$	$\{(-\infty, \infty)\}$

P

	PID	C	VT	RT
p_1	201	Spam filter	$[08/15, 08/24)$	$\{(-\infty, \infty)\}$
p_2	202	Spam filter	$[08/24, 08/27)$	$\{(-\infty, \infty)\}$

L

	Name	C	VT	RT
l_1	Ann	Spam filter	$[01/20, 08/18)$	$\{(-\infty, \infty)\}$
l_2	Bob	Spam filter	$[08/18, now)$	$\{(-\infty, \infty)\}$

Figure 2.1: Relations with ongoing time points.

Relation **B** illustrates bugs described by identifier *BID*, the name of the affected component *C*, the valid time interval VT during which the bug is open, and the reference time *RT* when the tuple belongs to the instantiated relations (cf. below and Section 2.7 for the details). For instance, tuple b_1 records deprioritized bug 500 for the Spam filter component that has been open from $01/25$ until *now*.

Relation **P** illustrates patches described by patch number *PID*, component *C* to which the patch applies, valid time interval *VT* during which the patch is live, and the reference time *RT*. For instance, tuple p_1 states that patch 201 of the Spam filter is live from $08/15$ until $08/24$ exclusively.

Relation **L** lists the technical leads. A technical lead is described by their name, component *C* they are responsible for, valid time interval *VT* during which they are responsible for the component, and the reference time *RT*. For instance, tuple l_2 records that Bob is the technical lead for the Spam filter component from $08/18$ until *now*.

Relations **B**, **P**, and **L** are *base ongoing relations*. All tuples belong to the instantiated relations at all reference times and have a trivial reference time, i.e., $RT = \{(-\infty, \infty)\}$. The reference time is restricted by predicates on ongoing attributes. We will discuss the restriction of a tuple's reference time in the following.

To schedule bug fixes, reprioritize bugs, and assess unresolved bugs, we run a query that joins bugs that affect the Spam filter with upcoming patches and technical leads:

$$\begin{aligned}\mathbf{V} \leftarrow \pi_{BID,\mathbf{B}.VT,PID,\text{Name},\mathbf{B}.VT \cap \mathbf{L}.VT}(\\ \sigma_{C=\text{'Spam filter'}}(\mathbf{B}) \\ \bowtie_{(\mathbf{B}.C=\mathbf{P}.C) \wedge (\mathbf{B}.VT \text{ before } \mathbf{P}.VT)} \mathbf{P} \\ \bowtie_{(\mathbf{B}.C=\mathbf{L}.C) \wedge (\mathbf{B}.VT \text{ overlaps } \mathbf{L}.VT)} \mathbf{L})\end{aligned}$$

We illustrate the computation of the reference time RT for $b_1 \bowtie_\theta p_1$ with $\theta = ((\mathbf{B}.C = \mathbf{P}.C) \wedge (\mathbf{B}.VT \text{ before } \mathbf{P}.VT))$. Conceptually, all occurrences of *now* in predicate $\theta(b_1, p_1)$ are replaced with each possible reference time *rt* in turn and the predicate is evaluated. This yields the following results for the *before* predicate:

rt	$[01/25, now)$	$[08/15, 08/24)$	$b_1.VT$ before $p_1.VT$
...	...	...	...
$08/14$	$[01/25, 08/14)$	$[08/15, 08/24)$	*true*
$08/15$	$[01/25, 08/15)$	$[08/15, 08/24)$	*true*
$08/16$	$[01/25, 08/16)$	$[08/15, 08/24)$	*false*
...	...	...	...

At all reference times when the join predicate evaluates to *true*, the result tuple belongs to the instantiated relations. In our example these are all reference times from $01/26$ up to $08/15$ and we get $RT = \{[01/26, 08/16)\}$.

Query result **V** includes the tuples illustrated in Figure 2.2. Note that (1) all ongoing time points are preserved in **V**. For instance, the value of the $\mathbf{B}.VT$ attribute makes it possible to identify prioritized and deprioritized bugs. (2) The intersection $\mathbf{B}.VT \cap \mathbf{L}.VT$ states when a technical lead is responsible for a bug. Consider tuple v_1 with $b_1.VT \cap l_1.VT = [01/25, +08/18)$, which is an ongoing time interval. Tuple v_1 states that Ann is the responsible technical lead for bug 500 from $01/25$ until possibly earlier, but not later than $08/17$. Clearly, fixed time points together with *now* are not sufficient to represent such results. (3) The reference time of a tuple is restricted

by predicates on ongoing attributes. For each operator, the reference time of the result tuples is determined by the reference times when the input tuples belong to the instantiated relations *and* the reference times when the predicate evaluates to *true*. The reference time of the input tuples is relevant since it is the result of predicates in earlier operators that derive these tuples. For instance, the reference time of the result tuples of join $\sigma_{C='\text{Spam filter}'}(\mathbf{B}) \bowtie_\theta \mathbf{P}$ was restricted by join predicate θ. These tuples are then input tuples for the join with ongoing relation **L**.

V

	BID	**B**.VT	PID	Name	**B**.VT∩**L**.VT	RT
v_1	500	$[01/25, now)$	201	Ann	$[01/25, +08/18)$	$\{[01/26, 08/16)\}$
v_2	500	$[01/25, now)$	202	Ann	$[01/25, +08/18)$	$\{[01/26, 08/25)\}$
v_3	500	$[01/25, now)$	202	Bob	$[08/18, now)$	$\{[08/19, 08/25)\}$
v_4	501	$[03/30, 08/21)$	202	Ann	$[03/30, 08/18)$	$\{(-\infty, \infty)\}$
v_5	501	$[03/30, 08/21)$	202	Bob	$[08/18, +08/21)$	$\{[08/19, \infty)\}$

Figure 2.2: Query result **V** remains valid as time passes by.

2.3 Related Work

The most commonly used ongoing time point is *now*. The state-of-the-art approach to deal with ongoing time points is to instantiate them, i.e., replace them with the reference time. Commercial database systems use the compile time as the reference time whereas research approaches usually use the evaluation time as the reference time. Below we discuss the implications of both choices for storing ongoing time points, query processing, and the validity of query results.

Existing database systems cannot store ongoing time points. They instantiate ongoing time points immediately at compile time when statements are issued. The SQL-92 standard [MS93] includes the reserved keywords CURRENT_TIME, CURRENT_DATE, and CURRENT_TIMESTAMP that denote the ongoing time point *now* for different time granularities. These constructs can be used in SQL statements, but are instantiated immediately at compile time.

Various research approaches have progressed the basic solution offered by commercial database systems. The key idea is to store ongoing time points and instantiate them when accessing the data during query processing. The advantage of instantiating ongoing time points is that existing query processing techniques can be used since the instantiation eliminates ongoing time points [TCG+93, SSJ94, LSM05, BGJ06, BBJS97, BJS00, DBGJ16]. The disadvantage is that query results are only valid at the chosen reference time and get outdated by time passing by.

Below we discuss different aspects of the instantiation that have been investigated [CDI+97, TJS00, FM96, JL01]. Throughout, we use $\mathscr{T}$ to denote the domain of fixed time points.

Clifford et al. [CDI+97, DJTS09] propose a solution that handles ongoing time point *now* during query processing. Their framework instantiates *now* whenever it is accessed. Thus, queries are evaluated on instantiated relations without ongoing time points. This yields result relations that are only valid at the time when *now* was accessed.

Anselma et al. [ASTS13] propose an algebra for relations with ongoing time points. Their goal is an approach that copes with four commonly used representations of *now*: *Min*, *Max* [TJB97], *Null*, and *Empty Range* [STS03, SST09]. Their time domain is $\mathscr{T} \cup \{now\}$. They introduce intersection and difference functions that may keep ongoing time points uninstantiated. For instance, ongoing time points are not instantiated when the resulting time interval contains *now* as end point like in $[10/14, now) \cap [10/17, now) = [10/17, now)$. Their approach must instantiate *now* for more complex end points. For instance, $[10/17, 10/22) \cap [10/17, now) = [10/17, 10/20)$ at reference time $10/20$. Anselma et al. [APS+16] have extended their approach to support indeterminacy for tuples with *now*. They have not worked out how predicates on ongoing time points are defined and evaluated.

Snodgrass [Sno87] proposes *Forever* instead of the ongoing time point *now*. *Forever* denotes the largest time point in the time domain, which is a fixed time point. Existing query evaluation approaches for relations without ongoing time points can be used on relations that use *Forever*. However, replacing *now* with *Forever* leads to incorrect results. For instance, at reference time $05/14$ the query *"Which bugs might be resolved before patch 201 goes live?"* is not answered correctly. Evaluating the query on relations **B** and **P** of Figure 2.1 with *Forever* as the end point results in bug 500 not being part of the result relation, which is not correct.

Torp et al. [TJS04] propose a solution for modifications of temporal databases. They show that performing temporal modifications on tuples that are instantiated when accessed leads to incorrect modifications and thus, incorrect data in the database. The authors propose time domain $\mathscr{T}_f = \mathscr{T} \cup \{\min(a, now) | a \in \mathscr{T}\} \cup \{\max(a, now) | a \in \mathscr{T}\}$ to handle such modifications. Instead of *now*, they use the minimum and maximum of a time point and *now* to correctly modify the database. Time domain T_f supports intersection and difference functions that do not instantiate ongoing time points. Torp et al. use these two functions to express temporal modifications that remain valid as time passes by. Their approach cannot evaluate predicates on uninstantiated time attributes. Queries with such predicates resort to Clifford's approach. Thus, query results get invalidated by time passing by.

Moving objects [GS05] change their spatial position as time advances. Research approaches in this area deal with different types of queries on moving objects: static queries [COTN08, TYJ09], continuous queries [ZQL+12, NATM15, LLBY14, HXL05], and time-parametrized queries [TP02]. *Static queries* instantiate the moving objects at a given reference time and are evaluated at fixed spatial positions. These approaches are similar to the approach of Clifford et al. [CDI+97], which instantiates ongoing time points. *Continuous queries* compute results that remain valid for a short time span, e.g., 10 seconds, before the query is re-evaluated. The results are continuously returned to applications. A query result contains pairs of moving object(s) and the reference times when the pair belongs to the result. Structurally, the query result is similar to ongoing relations with a reference time attribute. However, the result of a continuous query is only valid for a short time span and gets invalidated by time passing by. *Time-parametrized queries* [TP02] incrementally determine their results. The result consists of three parts: the objects that satisfy the spatial query, the reference time until when the result is valid, and the objects that change the result. The result is only valid from the time when the query was issued until the returned reference time.

Now-relative and indeterminate time points have been proposed as extensions of ongoing time point *now* [CDI+97]. A now-relative time point, e.g., $now + 5$ days, shifts *now* by 5 days into the future. An indeterminate time point specifies a period during which an event will occur. For instance, the indeterminate time point $04/17 \sim 04/20$ as the end point of a resolved bug states that the resolution occurred sometime between $04/17$ and $04/20$. These extensions are orthogonal to our generalization of *now*.

2.4 Preliminaries

We assume a linearly ordered, discrete *time domain* $\mathscr{T}$ with $-\infty$ as the lower limit and ∞ as the upper limit. A *time point* is an element of time domain $\mathscr{T}$. A *time interval* $[t_s, t_e)$ consists of an inclusive start point t_s and an exclusive end point t_e. *Fixed* data types consist of values that do not change as time passes by. Examples are integers, strings, booleans, and time points of $\mathscr{T}$. *Ongoing* data types include values that change as time passes by. Ongoing values can be *instantiated* to fixed values. We consider the following ongoing data types: ongoing time points, ongoing booleans, and composite structures (intervals, tuples, relations) that include ongoing time points. The bind operator $\|x\|_{rt}$ performs the instantiation of x at reference time $rt \in \mathscr{T}$. If x is composite each component is instantiated. We use the ^{F}-superscript for operations on

fixed data types. For instance, $\min^F$ is the standard minimum function over fixed arguments, i.e., $\min^F(j,k) = j$ if $j < k$ and $\min^F(j,k) = k$ otherwise.

$R = (\mathbf{A})$ denotes the schema of a fixed relation $\mathbf{R}$ with fixed attributes $\mathbf{A} = A_1, \ldots, A_n$. A tuple r with schema R is a finite list that contains for every A_i a value from the domain of A_i. A relation $\mathbf{R}$ over schema R is a finite set of tuples over R. $r.A_i$ denotes the value of attribute A_i in tuple r. $\theta(r)$ denotes the application of predicate θ to tuple r. An *ongoing relation* is a relation with fixed and ongoing attributes $\mathbf{A}$ and a reference time attribute RT (cf. Definition 5). The value of RT is a set of fixed time intervals.

Valid time [JCG+92], transaction time [JCG+92], and reference time are separate concepts. Consider a tuple b that refers to bug 500 with valid time $VT = [01/25, now)$, transaction time $TT = [01/26, now)$, and reference time $RT = \{[03/15, \infty)\}$. The valid time states when a tuple is valid in the real world: bug 500 is open from $01/25$ until *now*. The valid time is set by the user. The transaction time states when a tuple was modified in the relation: tuple b was inserted in $01/26$ and not modified since. The transaction time is restricted by the database system through database modifications, i.e., insert, update, and delete statements. The reference time states when a tuple belongs to the instantiated relations: tuple b belongs to the instantiated relations from $03/15$ on. The reference time is set by the database system and restricted by the predicates on ongoing attributes in queries.

2.5 Ongoing Time Data Types

This section defines the ongoing time domain Ω, ongoing time points, and ongoing time intervals. In contrast to previously proposed ongoing time domains, Ω is closed for minimum and maximum functions (cf. proof of Theorem 1).

2.5.1 Ongoing Time Points

Definition 1 (Ongoing Time Domain Ω)**.** *Let $\mathscr{T}$ be the time domain of fixed time points. Ongoing time domain Ω consists of all possible ongoing time points a+b:*

$$\Omega = \{a+b \mid \exists a,b \in \mathscr{T}(a \leq b)\}$$

The intuitive meaning of the ongoing time point a+b is *not earlier than a, but not later than b*. For instance, $10/17$+$10/19$ means *not earlier than* $10/17$*, but not later than* $10/19$.

Definition 2 (Ongoing Time Point)**.** *Let $rt \in \mathscr{T}$ be a reference time and $a,b \in \mathscr{T}$ with $a \leq b$. The ongoing time point a+b is defined as*

$$\|a{+}b\|_{rt} = \begin{cases} a & rt \leq a \\ rt & a < rt < b \\ b & otherwise \end{cases}$$

For instance, ongoing time point $10/17$+$10/19$ instantiates to time point $10/17$ up to reference time $10/17$. Between reference times $10/17$ and $10/19$ the ongoing time point instantiates to the reference time. Afterwards, it instantiates to time point $10/19$. This is illustrated in

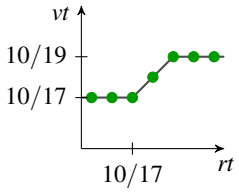


Figure 2.3: Illustration of ongoing time point $10/17$+$10/19$.

A *fixed* time point a, current time point *now*, a *growing* time point a+, and a *limited* time point +b can all be expressed as ongoing time points of the form a+b. This is illustrated in Figure 2.4. For instance, fixed time point $a = a$+a is an ongoing time point that instantiates to time point a at all reference times; time point $now = -\infty$+∞ is an ongoing time point that instantiates to the reference time at all reference times.

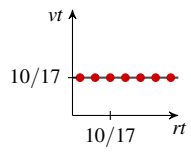
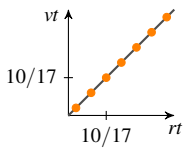
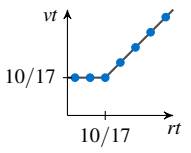
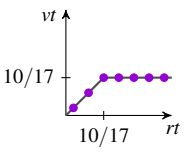
	Fixed time point	**Time point *now***	**Growing time point**	**Limited time point**
Notation				
- as a+b	a+a	$-\infty$+∞	a+∞	$-\infty$+b
- short	a	*now*	a+	+b
Meaning	time point a	the *current* time point	not earlier than a, possibly later	possibly earlier, but not later than b
Example	$10/17$	*now*	$10/17$+	+$10/17$
Semantics				

Figure 2.4: Ongoing time points expressed as a+b.

Table 2.1 summarizes the properties of time domains $\mathscr{T}$, $\mathscr{T}_{now} = \mathscr{T} \cup \{now\}$ [CDI+97], $\mathscr{T}_f = \mathscr{T} \cup \{\min(a, now) | \ a \in \mathscr{T}\} \cup \{\max(a, now) | \ a \in \mathscr{T}\}$ [TJS04], and Ω. For each time domain we show if it includes fixed or ongoing time points and if it is closed for min and max.

Table 2.1: Properties of time domains.

Time Domain	Fixed	Ongoing	Closed
$\mathscr{T}$	yes	no	yes
$\mathscr{T}_{now}$	yes	yes	no
$\mathscr{T}_f$	yes	yes	no
Ω	**yes**	**yes**	**yes**

2.5.2 Ongoing Time Intervals

An ongoing time interval $[t_s, t_e)$ is a closed-open time interval with domain $\Omega \times \Omega$. As an example, time interval $[10/17, now)$ is an ongoing time interval. An ongoing time interval can be instantiated to a fixed time interval by instantiating start and end points:

$$\forall rt \in \mathscr{T}(\|[t_s, t_e)\|_{rt} = [\|t_s\|_{rt}, \|t_e\|_{rt}))$$

The ongoing time interval $[a{+}b, c{+}d)$ generalizes *fixed* time intervals, *expanding* time intervals, and *shrinking* time intervals. Their semantics are illustrated in Figure 2.5. For instance, an expanding time interval instantiates to time intervals whose duration increases with increasing reference time. The duration can increase for all reference times or up to a certain reference time. An example for the first case is ongoing time interval $[10/17, now)$ with $d = \infty$. An example for the latter case is ongoing time interval $[10/17, 10/19{+}10/21)$ with $d = 10/21$. It instantiates to time intervals with increasing duration up to reference time $10/21$. From reference time $10/21$ on, it instantiates to time interval $[10/17, 10/21)$.

An ongoing time interval can be partially empty. A partially empty time interval instantiates to empty time intervals at some reference times and to non-empty time intervals at others. This is illustrated in Figure 2.5. For instance, ongoing time interval $[10/17, now)$ instantiates to empty time intervals up to reference time $10/17$. At these reference times, end point *now* instantiates to time points that are less than or equal to start point $10/17$ and the interval is empty. Afterwards, *now* instantiates to time points greater than $10/17$ and $[10/17, now)$ instantiates to non-empty time intervals.

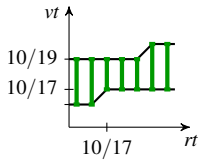
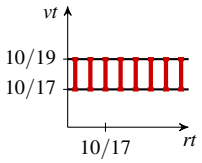
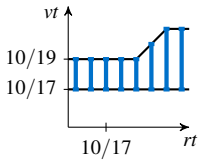
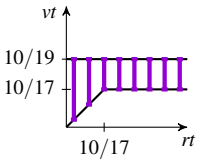
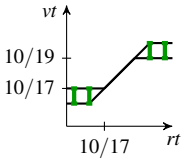
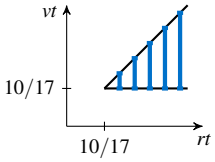
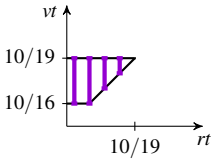
	Ongoing time interval	Type: Fixed time interval	Type: Expanding time interval	Type: Shrinking time interval
non-empty	if $a \le b < c \le d$	if $a = b < c = d$	if $a = b < c < d$	if $a < b < c = d$
	$[10/16+10/17,$ $10/19+10/20)$	$[10/17, 10/19)$	$[10/17, 10/19+10/21)$	$[+10/17, 10/19)$
				
partially empty	if $a < c \le b$ or $c \le a \le b < d$	never	if $c \le a \le b < d$	if $a < c \le d \le b$
	$[10/16+10/19,$ $10/17+10/20)$	–	$[10/17, now)$	$[10/16+, 10/19)$
				

Figure 2.5: Illustration of ongoing time intervals $[a+b, c+d)$.

2.6 Operations on Ongoing Data Types

This section defines operations, i.e., functions, predicates, and logical connectives, on ongoing time data types whose results remain valid as time passes by. At each reference time, their results are equal to the results obtained by the corresponding operation on fixed data types. We provide and prove equivalences for our six core operations $<, \min, \max, \wedge, \vee, \neg$ and show how we use these core operations in equivalences for additional operations on ongoing data types.

Since ongoing time points and time intervals instantiate to different values depending on the reference time the truth value of predicates depends on the reference time. To represent their result, we use *ongoing booleans* whose boolean value depends on the reference time.

Definition 3 (Ongoing Boolean)**.** *Let $rt \in \mathscr{T}$ be a reference time. Let $S_t \subseteq \mathscr{T}$ and $S_f \subseteq \mathscr{T}$ be disjoint subsets of all possible reference times with $S_t \cup S_f = \mathscr{T}$. The ongoing boolean $\mathbf{b}[S_t, S_f]$ is defined as*

$$\|\mathbf{b}[S_t, S_f]\|_{rt} = \begin{cases} true & rt \in S_t \\ false & rt \in S_f \end{cases}$$

An ongoing boolean $\mathbf{b}[S_t, S_f]$ is *true* at the reference times in S_t and *false* at the reference times in S_f. For instance, ongoing boolean $\mathbf{b}[\{[10/18, \infty)\}, \{(-\infty, 10/18)\}]$ is *true* at reference time $10/18$ (as well as at all later reference times), and it is *false* at the reference times earlier than $10/18$. Ongoing booleans generalize booleans. Boolean *true* is equivalent to ongoing boolean $\mathbf{b}[\{(-\infty, \infty)\}, \emptyset]$, which is *true* at all reference times. Boolean *false* is equivalent to ongoing boolean $\mathbf{b}[\emptyset, \{(-\infty, \infty)\}]$. The generalization makes it possible to combine predicates that evaluate to booleans with predicates that evaluate to ongoing booleans in logical expressions.

Definition 4 (Core Operations)**.** *Let $t_1, t_2, t \in \Omega$ be ongoing time points. Let $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b} \in \Gamma$ be ongoing booleans. The core operations on ongoing data types are defined as follows:*

Operation	***Definition***
$<$	$t_1 < t_2 = \mathbf{b}$ *iff* $\forall rt \in \mathscr{T}(\|\mathbf{b}\|_{rt} \Leftrightarrow \|t_1\|_{rt} <^F \|t_2\|_{rt})$
min	$\min(t_1, t_2) = t$ *iff* $\forall rt \in \mathscr{T}(\|t\|_{rt} = \min^F(\|t_1\|_{rt}, \|t_2\|_{rt}))$
max	$\max(t_1, t_2) = t$ *iff* $\forall rt \in \mathscr{T}(\|t\|_{rt} = \max^F(\|t_1\|_{rt}, \|t_2\|_{rt}))$
$\wedge$	$\mathbf{b}_1 \wedge \mathbf{b}_2 = \mathbf{b}$ *iff* $\forall rt \in \mathscr{T}(\|\mathbf{b}\|_{rt} \Leftrightarrow \|\mathbf{b}_1\|_{rt} \wedge^F \|\mathbf{b}_2\|_{rt})$
$\vee$	$\mathbf{b}_1 \vee \mathbf{b}_2 = \mathbf{b}$ *iff* $\forall rt \in \mathscr{T}(\|\mathbf{b}\|_{rt} \Leftrightarrow \|\mathbf{b}_1\|_{rt} \vee^F \|\mathbf{b}_2\|_{rt})$
$\neg$	$\neg\mathbf{b}_1 = \mathbf{b}$ *iff* $\forall rt \in \mathscr{T}(\|\mathbf{b}\|_{rt} \Leftrightarrow \neg^F \|\mathbf{b}_1\|_{rt})$

An operation on ongoing data types evaluates to a result that, at each reference time, is equal to the result obtained by the corresponding operation on fixed data types. This yields results that remain valid as time passes by.

All other predicates and functions on ongoing data types are defined analogously.

Example 9. *Consider* min *for ongoing time points and the corresponding function* $\min^F$ *for fixed time points. The result of* $\min(10/17, now)$ *is ongoing time point* $t = +10/17$ *(cf. Theorem 1). At each reference time, it is equal to the time point obtained from evaluating* $\min^F$ *on the instantiated input arguments, i.e.,* $+10/17$ *is equal to* $\min^F(\|10/17\|_{rt}, \|now\|_{rt})$ *at every reference time rt. Figure 2.6 illustrates the equality for reference times* $10/15$ *and* $10/19$.

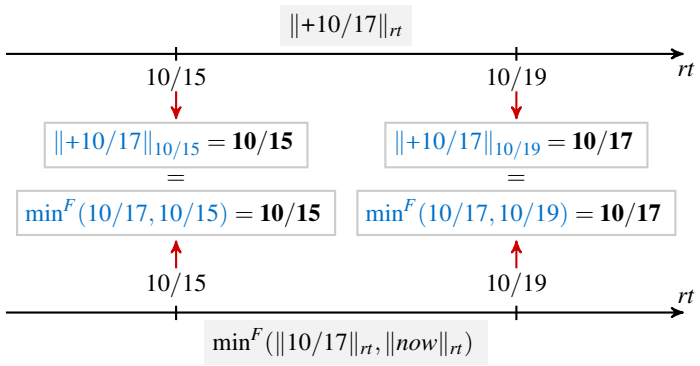


Figure 2.6: The result of min remains valid.

Theorem 1. *Let $a{+}b, c{+}d \in \Omega$ be ongoing time points. Let $\mathbf{b}[S_t, S_f], \mathbf{b}[\tilde{S}_t, \tilde{S}_f] \in \Gamma$ be ongoing booleans. The results of the operations on ongoing data types given in Definition 4 are equivalent to the following ongoing values:*

Operation	***Equivalence***		
$<$	$a{+}b < c{+}d \equiv$		
	$\mathbf{b}[\{(-\infty,\infty)\},\emptyset]$	$a \le b < c \le d$	(1)
	$\mathbf{b}[\{(-\infty,c)\},\{[c,\infty)\}]$	$a < c \le d \le b$	(2)
	$\mathbf{b}[\{[b+1,\infty)\},\{(-\infty,b+1)\}]$	$c \le a \le b < d$	(3)
	$\mathbf{b}[\{(-\infty,c),[b+1,\infty)\},\{[c,b+1)\}$	$a < c \le b < d$	(4)
	$\mathbf{b}[\emptyset,\{(-\infty,\infty)\}]$	*otherwise*	(5)
min	$\min(a{+}b, c{+}d) \equiv \min^F(a,c){+}\min^F(b,d)$		
max	$\max(a{+}b, c{+}d) \equiv \max^F(a,c){+}\max^F(b,d)$		
$\wedge$	$\mathbf{b}[S_t,S_f] \wedge \mathbf{b}[\tilde{S}_t,\tilde{S}_f] \equiv \mathbf{b}[S_t \cap^F \tilde{S}_t, S_f \cup^F \tilde{S}_f]$		
$\vee$	$\mathbf{b}[S_t,S_f] \vee \mathbf{b}[\tilde{S}_t,\tilde{S}_f] \equiv \mathbf{b}[S_t \cup^F \tilde{S}_t, S_f \cap^F \tilde{S}_f]$		
$\neg$	$\neg\mathbf{b}[S_t,S_f] \equiv \mathbf{b}[S_f,S_t]$		

Proof. We prove the equivalences in the order provided in Theorem 1.

The equivalence for $a{+}b < c{+}d$ is proven by showing for each ordering of a, b, c, and d that the definition of $<$ holds (cf. Definition 4). We show for the ordering $a < c = d < b$ that ongoing boolean $\mathbf{b}[\{(-\infty,c)\},\{[c,\infty)\}]$ (case 2 of the equivalence) fulfills the definition, i.e., $\forall rt \in \mathscr{T}(\|\mathbf{b}[\{(-\infty,c)\},\{[c,\infty)\}]\|_{rt} \Leftrightarrow \|a{+}b\|_{rt} <^F \|c{+}d\|_{rt})$. Table 2.2 shows that the definition is fulfilled at every reference time: $<^F$ evaluates to the same boolean as ongoing boolean

$\mathbf{b} = \mathbf{b}[\{(-\infty, c)\}, \{[c, \infty)\}]$ instantiates to. The equivalence is proven for the other orderings analogously.

Table 2.2: The equivalence fulfills the definition of $<$.

rt	$\|a+b\|_{rt}$	$\|c+d\|_{rt}$	$<^F$	$\|\mathbf{b}\|_{rt}$
$\mathbf{rt} \leq a < c = d < b$	a	c	true	true
$a < \mathbf{rt} < c = d < b$	rt	c	true	true
$a < \mathbf{rt} = c = d < b$	rt	c	false	false
$a < c = d < \mathbf{rt} < b$	rt	c	false	false
$a < c = d < b \leq \mathbf{rt}$	b	c	false	false

We prove $\min(a+b, c+d) \equiv \min^F(a,c)+\min^F(b,d)$ by showing that (1) $\min^F(a,c)+\min^F(b,d)$ is an ongoing time point of Ω, and (2) the definition of min (cf. Definition 4) holds for $t = \min^F(a, c)+\min^F(b,d)$. Let $a+b, c+d \in \Omega$ be two ongoing time points with $a \leq b$ and $c \leq d$. First, for fixed values, $\min^F(a,c) \leq a$ and $\min^F(a,c) \leq c$ hold. To prove $\min^F(a,c)+\min^F(b,d) \in \Omega$ we must show that $\min^F(a,c) \leq \min^F(b,d)$.

$$\begin{aligned}
&\text{Case 1: } \min^F(b,d) = b \\
&\qquad \min^F(a,c) \leq a \leq b = \min^F(b,d) \\
&\text{Case 2: } \min^F(b,d) = d \\
&\qquad \min^F(a,c) \leq c \leq d = \min^F(b,d)
\end{aligned}$$

Second, we show that the definition of min holds for $t = \min^F(a,c)+\min^F(b,d)$. Let $rt \in \mathscr{T}$ be a reference time. From Definition 2 it follows that the instantiation $\|a+b\|_{rt}$ is equivalent to $\min^F(b, \max^F(a, rt))$.

$$\begin{aligned}
&\|t\|_{rt} = \min^F(\|a+b\|_{rt}, \|c+d\|_{rt}) \\
&\Leftrightarrow \ \|\min^F(a,c)+\min^F(b,d)\|_{rt} \\
&\qquad = \min^F(\min^F(b, \max^F(a,rt)), \min^F(d, \max^F(c,rt))) \\
&\Leftrightarrow^1 \ \min^F(\min^F(b,d), \max^F(\min^F(a,c), rt)) \\
&\qquad = \min^F(\min^F(b,d), \min^F(\max^F(a,rt), \max^F(c,rt))) \\
&\Leftrightarrow^2 \ \min^F(\min^F(b,d), \max^F(\min^F(a,c), rt)) \\
&\qquad = \min^F(\min^F(b,d), \max^F(\min^F(a,c), rt))
\end{aligned}$$

[1] $\min^F$ is associative, i.e., $\min^F(\min^F(w,x),\min^F(y,z)) = \min^F(\min^F(w,y),\min^F(x,z))$
[2] $\min^F$ and $\max^F$ are distributive over each other, i.e., $\min^F(\max^F(x,z),\max^F(y,z)) = \max^F(\min^F(x,y),z)$

Thus, $\min(a{+}b, c{+}d) \equiv \min^F(a,c){+}\min^F(b,d)$ holds. The equivalence of max is proven analogously.

The logical conjunction of two ongoing booleans is ongoing boolean $\mathbf{b}[S_t \cap^F \tilde{S}_t, S_f \cup^F \tilde{S}_f]$. It instantiates to *true* at the reference times when both input ongoing booleans instantiate to *true*, i.e., $S_t \cap^F \tilde{S}_t$; it instantiates to *false* when at least one of the input ongoing booleans instantiate to *false*, i.e., at the union $S_f \cup^F \tilde{S}_f$. The disjunction of two ongoing booleans is ongoing boolean $\mathbf{b}[S_t \cup^F \tilde{S}_t, S_f \cap^F \tilde{S}_f]$. It instantiates to *true* at the reference times when at least one of the input ongoing booleans instantiates to *true*. The negation of an ongoing boolean $\mathbf{b}[S_t, S_f]$ is $\mathbf{b}[S_f, S_t]$. This means that at the reference times when the input ongoing boolean instantiates to *true*, the resulting ongoing boolean instantiates to *false*. □

We use our core operations to provide equivalences for predicates and functions on ongoing time points and time intervals. Table 2.3 illustrates the equivalences for selected predicates and functions. For instance, the intersection $[t_s,t_e) \cap [\tilde{t}_s,\tilde{t}_e)$ on ongoing time intervals is equivalent to the ongoing time interval $[\max(t_s,\tilde{t}_s), \min(t_e,\tilde{t}_e))$.

For predicates on ongoing time intervals we must explicitly consider the non-emptiness of the ongoing time intervals. For instance, the overlaps predicate is equivalent to the ongoing boolean that results from the usual overlaps check $t_s < \tilde{t}_e \ \wedge \ \tilde{t}_s < t_e$ and an explicit non-empty check $t_s < t_e \wedge \tilde{t}_s < \tilde{t}_e$. The explicit non-empty check is necessary because ongoing time intervals can be partially empty. It is not sufficient to check if the ongoing input time intervals are not empty at all reference times; we must check non-emptiness at each reference time.

Example 10. *Consider the overlaps predicate. At all reference times when one of the input time intervals instantiates to an empty time interval, the non-empty check ensures that the predicate evaluates to* false. *At all other reference times, the overlaps check determines the result. At reference time* $10/16$*, ongoing time interval* $[10/17, now)$ *instantiates to an empty time interval and thus, predicate* $[10/17, now)$ *overlaps* $[10/14, 10/20)$ *evaluates to* false. *At reference time* $10/18$*, both ongoing input time intervals instantiate to non-empty time intervals and the overlaps check evaluates to* true. *Thus, predicate* $[10/17, now)$ *overlaps* $[10/14, 10/20)$ *evaluates to ongoing boolean* $\mathbf{b}[\{[10/18, \infty)\}, \{(-\infty, 10/18)\}]$.

Table 2.3: Equivalences for predicates and function on ongoing time points and time intervals.

Operation	Equivalence
$\leq$	$t_1 \leq t_2 \equiv \neg(t_2 < t_1)$
	Example $now \leq 10/17$
	$= \neg(\mathbf{b}[\{[10/18,\infty)\},\{(-\infty,10/18)\}])$
	$= \mathbf{b}[\{(-\infty,10/18)\},\{[10/18,\infty)\}]$
$=$	$t_1 = t_2 \equiv t_1 \leq t_2 \wedge t_2 \leq t_1$
	Example $(10/17 = now)$
	$= \mathbf{b}[\{[10/17,10/18)\},\{(-\infty,10/17),[10/18,\infty)\}]$
$\neq$	$t_1 \neq t_2 \equiv (t_1 < t_2) \vee (t_2 < t_1)$
	Example $10/17 \neq now$
	$= \mathbf{b}[\{(-\infty,10/17),[10/18,\infty)\},\{[10/17,10/18)\}]$
before	$[t_s,t_e)$ before $[\tilde{t}_s,\tilde{t}_e) \equiv t_e \leq \tilde{t}_s \wedge t_s < t_e \wedge \tilde{t}_s < \tilde{t}_e$
	Example $[10/17,now)$ before $[10/20,10/25)$
	$= \mathbf{b}[\{[10/18,10/21)\},\{(-\infty,10/18),[10/21,\infty)\}]$
meets	$[t_s,t_e)$ meets $[\tilde{t}_s,\tilde{t}_e) \equiv t_e = \tilde{t}_s \wedge t_s < t_e \wedge \tilde{t}_s < \tilde{t}_e$
	Example $[10/17,now)$ meets $[10/20,10/25)$
	$= \mathbf{b}[\{[10/20,10/21)\},\{(-\infty,10/20),[10/21,\infty)\}]$
overlaps	$[t_s,t_e)$ overlaps $[\tilde{t}_s,\tilde{t}_e) \equiv t_s < \tilde{t}_e \wedge \tilde{t}_s < t_e \wedge t_s < t_e \wedge \tilde{t}_s < \tilde{t}_e$
	Example $[10/17,now)$ overlaps $[10/14,10/20)$
	$= \mathbf{b}[\{[10/18,\infty)\},\{(-\infty,10/18)\}]$
starts	$[t_s,t_e)$ starts $[\tilde{t}_s,\tilde{t}_e) \equiv t_s = \tilde{t}_s \wedge t_s < t_e \wedge \tilde{t}_s < \tilde{t}_e$
	Example $[10/17,now)$ starts $[10/17,10/20)$
	$= \mathbf{b}[\{[10/18,\infty)\},\{(-\infty,10/18)\}]$
finishes	$[t_s,t_e)$ finishes $[\tilde{t}_s,\tilde{t}_e) \equiv t_e = \tilde{t}_e \wedge t_s < t_e \wedge \tilde{t}_s < \tilde{t}_e$
	Example $[10/17,now)$ finishes $[10/20,10/25)$
	$= \mathbf{b}[\{[10/25,10/26)\},\{(-\infty,10/25),[10/26,\infty)\}]$
during	$[t_s,t_e)$ during $[\tilde{t}_s,\tilde{t}_e) \equiv\ (\tilde{t}_s \leq t_s \wedge t_e \leq \tilde{t}_e \wedge t_s < t_e \wedge \tilde{t}_s < \tilde{t}_e) \vee (t_e \leq t_s \wedge \tilde{t}_s < \tilde{t}_e)$
	Example $[10/20,10/25)$ during $[10/17,now)$
	$= \mathbf{b}[\{[10/25,\infty)\},\{(-\infty,10/25)\}]$
equals	$[t_s,t_e)$ equals $[\tilde{t}_s,\tilde{t}_e) \equiv\ (t_s = \tilde{t}_s \wedge t_e = \tilde{t}_e \wedge t_s < t_e \wedge \tilde{t}_s < \tilde{t}_e) \vee (t_e \leq t_s \wedge \tilde{t}_e \leq \tilde{t}_s)$
	Example $[10/17,now)$ equals $[10/17,10/20)$
	$= \mathbf{b}[\{[10/20,10/21)\},\{(-\infty,10/20),[10/21,\infty)\}]$
$\cap$	$[t_s,t_e) \cap [\tilde{t}_s,\tilde{t}_e) \equiv [\max(t_s,\tilde{t}_s),\min(t_e,\tilde{t}_e))$
	Example $[10/17,now) \cap [10/14,10/20) = [10/17,+10/20)$

2.7 Relational Algebra

The first subsection introduces *ongoing relations* to represent query results that remain valid at varying times. Ongoing relations include tuples that belong to instantiated relations at some reference times only. An ongoing relation models this by associating each tuple with a reference time attribute. The value of the reference time attribute is restricted by the predicates on ongoing attributes. The second subsection defines the operators of the relational algebra as operators on ongoing relations.

2.7.1 Ongoing Relations

Definition 5 (Schema of an Ongoing Relation)**.** *Let* $\mathbf{A}$ *be a list of fixed and ongoing attributes* $A_1, \ldots, A_n$ *and* RT *be the reference time attribute. Then,*

$$R = (\mathbf{A}, RT)$$

is the schema of an ongoing relation.

A tuple belongs to the instantiated relations at the reference times that are contained in the value of the tuple's reference time attribute RT. In a base tuple, the value of RT is set to trivial reference times, i.e., $RT = \{(-\infty, \infty)\}$, by the database system. The reference time of tuples is then restricted by predicates on ongoing attributes.

The bind operator $\|\mathbf{R}\|_{rt}$ instantiates an ongoing relation $\mathbf{R}$ at reference time $rt \in \mathscr{T}$ by instantiating the ongoing attributes of each tuple at reference time rt. It omits tuples whose reference time RT does not contain rt:

$$\|\mathbf{R}\|_{rt} = \{x | \exists r \in \mathbf{R}(x.\mathbf{A} = \|r.\mathbf{A}\|_{rt} \wedge rt \in r.RT)\}$$

2.7.2 Operators on Ongoing Relations

The definition of the relational algebra operators on ongoing relations follows the approach in Definition 4. For instance, selection $\sigma_\theta(\mathbf{R})$ for ongoing relations is defined as

$$\sigma_\theta(\mathbf{R}) = \mathbf{V} \text{ iff } \forall rt \in \mathscr{T}(\|\mathbf{V}\|_{rt} \equiv \sigma^F_{\theta^F}(\|\mathbf{R}\|_{rt}))$$

Derived relational algebra operators are defined as usual. As an example, $\mathbf{R} \bowtie_\theta \mathbf{S} = \sigma_\theta(\mathbf{R} \times \mathbf{S})$.

Theorem 2. *Let* $\mathbf{R}, \mathbf{S}$ *be two ongoing relations with attributes* $\mathbf{A}$ *and* $\mathbf{C}$*, respectively. Let* $\mathbf{B} \subseteq \mathbf{A}$ *be a subset of the attributes of* $\mathbf{R}$ *and let predicate* θ *be composed of operations whose results remain valid as time passes by (cf. Section 2.6). The results of the relational algebra operators on ongoing relations are equivalent to the following ongoing relations:*

Operator	***Equivalence***
Projection	$\pi_{\mathbf{B}}(\mathbf{R}) \equiv \{x \mid \exists r \in \mathbf{R}(x.\mathbf{B} = r.\mathbf{B} \wedge x.RT = r.RT)\}$
Selection	$\sigma_\theta(\mathbf{R}) \equiv \{x \mid \exists r \in \mathbf{R}(x.\mathbf{A} = r.\mathbf{A} \wedge x.RT = (r.RT \wedge \theta(r)) \wedge x.RT \neq \emptyset)\}$
Cartesian product	$\mathbf{R} \times \mathbf{S} \equiv \{x \mid \exists r \in \mathbf{R}, s \in \mathbf{S}(x.\mathbf{A} = r.\mathbf{A} \wedge x.\mathbf{C} = s.\mathbf{C}$ $\wedge x.RT = (r.RT \wedge s.RT) \wedge x.RT \neq \emptyset)\}$
Union	$\mathbf{R} \cup \mathbf{S} \equiv \{x \mid x \in \mathbf{R} \vee x \in \mathbf{S}\}$
Difference	$\mathbf{R} - \mathbf{S} \equiv \{x \mid \exists r \in \mathbf{R}(x.\mathbf{A} = r.\mathbf{A} \wedge x.RT \neq \emptyset$ $\wedge x.RT = \{rt \in r.RT \mid \nexists s \in \mathbf{S}(\lVert r.\mathbf{A} \rVert_{rt} =^F \lVert s.\mathbf{A} \rVert_{rt} \wedge rt \in s.RT)\})\}$

Proof. We prove the equivalence for selection $\sigma_\theta(\mathbf{R})$. For the other operators, similar transformations from the reference time of result tuples to instantiated relations hold.

Let $\mathbf{R}$ be an ongoing relation and θ be a predicate with operations whose results remain valid as time passes by. Let σ^F be the selection for fixed relations and predicate θ^F be the predicate we get by replacing operations in θ with the corresponding fixed operations. We prove that $\lVert \mathbf{V} \rVert_{rt} = \sigma^F_{\theta^F}(\lVert \mathbf{R} \rVert_{rt})$ holds at all $rt \in \mathscr{T}$ for $\mathbf{V} = \{x \mid \exists r \in \mathbf{R}(x.\mathbf{A} = r.\mathbf{A} \wedge x.RT = (r.RT \wedge \theta(r)) \wedge x.RT \neq \emptyset)\}$.

$$
\begin{aligned}
&\lVert \mathbf{V} \rVert_{rt} = \sigma^F_{\theta^F}(\lVert \mathbf{R} \rVert_{rt}) \\
&\Leftrightarrow \lVert \{x \mid \exists r \in \mathbf{R}(x.\mathbf{A} = r.\mathbf{A} \wedge x.RT = (r.RT \wedge \theta(r)) \wedge x.RT \neq \emptyset)\} \rVert_{rt} \\
&\quad = \{u \mid u \in \lVert \mathbf{R} \rVert_{rt} \wedge \theta^F(u)\} \\
&\Rightarrow^* \{u \mid \exists r \in \mathbf{R}(u.\mathbf{A} = \lVert r.\mathbf{A} \rVert_{rt} \wedge rt \in (r.RT \wedge \theta(r)))\} \\
&\quad = \{u \mid \exists r \in \mathbf{R}(u.\mathbf{A} = \lVert r.\mathbf{A} \rVert_{rt} \wedge rt \in r.RT \wedge \theta^F(\lVert r \rVert_{rt}))\} \\
&\Leftrightarrow \{u \mid \exists r \in \mathbf{R}(u.\mathbf{A} = \lVert r.\mathbf{A} \rVert_{rt} \wedge rt \in r.RT \wedge \lVert \theta(r) \rVert_{rt})\} \\
&\quad = \{u \mid \exists r \in \mathbf{R}(u.\mathbf{A} = \lVert r.\mathbf{A} \rVert_{rt} \wedge rt \in r.RT \wedge \lVert \theta(r) \rVert_{rt})\}
\end{aligned}
$$

* The bind operator eliminates tuples with an empty reference time and thus ensures $RT \neq \emptyset$. □

As an example, selection $\sigma_\theta(\mathbf{R})$ selects a tuple $r \in \mathbf{R}$ by restricting the tuple's reference time RT. The reference time of the tuple is set to $r.RT \wedge \theta(r)$, i.e., the intersection of the reference time of the original tuple ($r.RT$) and the reference times when predicate $\theta(r)$ is satisfied. To restrict RT with an ongoing boolean, we convert a tuple's reference time into the set S_t of an ongoing boolean $\mathbf{b}[S_t, S_f]$ and calculate the conjunction between the ongoing booleans to determine the reference time of the result tuple.

Example 11. *Consider ongoing relation* $\mathbf{X}$ *with tuple* $x = (500, \textit{Spam filter}, [01/25, \textit{now}), \{(-\infty, 08/16)\})$ *and selection* $Q = \sigma_\theta(\mathbf{X})$ *with* $\theta = VT$ *overlaps* $[01/20, 08/18)$. *Query* Q *selects input tuple* x *at the reference times when it belongs to the instantiated input relations (up to reference time* $08/15$*)* and *when predicate* $\theta(x)$ *evaluates to* true. *The result of predicate* $\theta(x)$ *is ongoing boolean* $\mathbf{b}[\{[01/26, \infty)\}, \{(-\infty, 01/26)\}]$. *The reference time of result tuple* y *is* $x.RT \wedge \theta(x)$*:*

$$
\begin{aligned}
y.RT &= x.RT \wedge \theta(x) \\
&= \{(-\infty, 08/16)\} \wedge \mathbf{b}[\{[01/26, \infty)\}, \{(-\infty, 01/26)\}] \\
&= \mathbf{b}[\{(-\infty, 08/16)\}, \{[08/16, \infty)\}] \wedge \mathbf{b}[\{[01/26, \infty)\}, \{(-\infty, 01/26)\}] \\
&= \mathbf{b}[\{[01/26, 08/16)\}, \{(-\infty, 01/26), [08/16, \infty)\}] \\
&= \{[01/26, 08/16)\}
\end{aligned}
$$

Thus, for selection Q *on input tuple* x *we get result tuple:*

$$y = (500, \textit{Spam filter}, [01/25, \textit{now}), \{[01/26, 08/16)\})$$

Predicates on fixed attributes retain their standard behavior. If a predicate on fixed attributes evaluates to *true*, the result tuple's reference time does not change as it is restricted by the conjunction with ongoing boolean $\mathbf{b}[\{(-\infty, \infty)\}, \emptyset]$ ($\equiv$ *true*). If a predicate evaluates to *false*, the result tuple is omitted as the conjunction with ongoing boolean $\mathbf{b}[\emptyset, \{(-\infty, \infty)\}]$ ($\equiv$ *false*) results in an empty reference time.

2.8 Implementation

This section describes the implementation of ongoing data types in the kernel of PostgreSQL. Our implementation is space-efficient and optimized for evaluating the operations in Section 2.6.

Ongoing Time Data Types Our implementation supports ongoing time points with the two granularities offered by PostgreSQL: *dates* with a granularity of days and *timestamps* with a granularity of microseconds. The PostgreSQL *date* and *timestamp* data types are extended to structures composed of two fixed dates and two fixed timestamps, respectively, to represent ongoing time points a+b. Time point *now* is represented as $-\infty$+∞. Note that PostgreSQL natively provides representations for $-\infty$ and ∞ as fixed dates and timestamps. The extensions of the *date* and *timestamp* data types also yield support for ongoing time intervals of $\Omega \times \Omega$ as *daterange*s and *tsrange*s in PostgreSQL.

Reference Time RT We represent a tuple's reference time as a list of fixed time intervals. For the list, we use the built-in, variable-length data type *array* to leverage the built-in storage, indexing, and fetching mechanisms for variable length data types. Its variable length guarantees that PostgreSQL allocates the minimal amount of space to store the list of reference time intervals.

Ongoing Booleans We represent an ongoing boolean $\mathbf{b}[S_t, S_f] \in \Gamma$ with the set S_t of reference times when the ongoing boolean is *true*. S_t is represented with the same data type as a tuple's reference time. This is beneficial when restricting a tuple's reference time: the logical conjunction of a predicate and the tuple's reference time can then be directly computed (cf. Section 2.7.2). The time intervals used for S_t are maximal, non-overlapping, and sorted in ascending order. These properties yield an efficient implementation of the logical connectives with a sweep-line algorithm (cf. Algorithm 1).

We developed new algorithms for $<$, $\wedge$, $\vee$, and $\neg$. The less-than predicate minimizes the number of value comparisons and the implementation of the logical connectives processes each time interval just once. The other operations are implemented with the equivalences in Section 2.6.

Less-Than Predicate The less-than predicate for ongoing time points is implemented according to the case distinction in Theorem 1. The result of the less-than predicate is an ongoing boolean, which we represent as an array of time intervals for S_t as described above. Since an ongoing time point a+b ensures $a \leq b$, we use the decision tree in Figure 2.7 to determine the correct case with at most three comparisons.

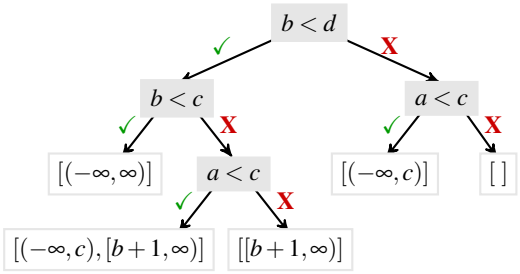


Figure 2.7: Decision tree for a+$b < c$+d.

Logical Connectives We use a sweep-line algorithm to implement the logical connectives. The implementation requires and guarantees arrays with non-overlapping time intervals that are sorted in ascending order. The implementation has the following three properties that make it efficient: 1. no sorting is required since a sweep-line algorithm guarantees sorted results at no cost, 2. each time interval of the input ongoing booleans is processed at most once, which minimizes the number of time intervals to be compared, and 3. the implementation minimizes the overall number of time point comparisons. Note that the logical connectives are not only used in predicates but also to calculate a tuple's reference time in a relational algebra operator (cf. Theorem 2). Algorithm 1 shows the implementation of the logical conjunction. The efficient implementation of the conjunction is important since the conjunction is used to calculate a result tuple's reference time in all relational algebra operators.

Procedure: Conjunction $\mathbf{b_1} \wedge \mathbf{b_2}$
Input: $\mathbf{b_1}, \mathbf{b_2} \in \Gamma$: two arrays of non-overlapping time intervals in ascending order
Output: $\mathbf{b_r} \in \Gamma$: array of non-overlapping time intervals in ascending order

1 $\mathbf{b_r} = [\,]$; $i_1 \leftarrow \mathbf{b_1}$.first; $i_2 \leftarrow \mathbf{b_2}$.first;
2 **while** $i_1 \neq$ nil $\wedge\, i_2 \neq$ nil **do**
3 **if** $i_1.t_e \leq i_2.t_s$ **then** $i_1 \leftarrow \mathbf{b_1}$.next;
4 **else if** $i_2.t_e \leq i_1.t_s$ **then** $i_2 \leftarrow \mathbf{b_2}$.next;
5 **else**
  `// append intersection of` i_1 `and` i_2
6 $\mathbf{b_r}$.append($[\max(i_1.t_s, i_2.t_s), \min(i_1.t_e, i_2.t_e)]$);
7 **if** $i_1.t_e < i_2.t_e$ **then** $i_1 \leftarrow \mathbf{b_1}$.next **else** $i_2 \leftarrow \mathbf{b_2}$.next;
8 **end**
9 **end**
10 **return** $\mathbf{b_r}$;

Algorithm 1: Conjunction on ongoing booleans.

Query Optimization For the relational operators on ongoing relations, the same rules hold as for the relational algebra operators on fixed relations. For instance, the equivalence $\sigma_{\theta_1 \wedge \theta_2}(\mathbf{R}) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(\mathbf{R}))$ holds for an ongoing relation $\mathbf{R}$. After the rewriting, existing optimization techniques, such as selection push-down, join ordering, and cost-based selection of evaluation algorithms, can be used.

To leverage database optimization strategies and algorithms for queries on ongoing relations, we split a conjunctive predicate into a conjunctive predicate over fixed attributes only and a conjunctive predicate that references ongoing attributes. The predicate over fixed attributes does not depend on the reference time and can therefore be evaluated in the where clause. The predicate over ongoing attributes is used in the calculation of the result tuple's reference time (cf. Theorem 2).

2.9 Evaluation

This section compares runtime, result size, and storage requirements of our solution with the state-of-the-art solution from Clifford et al. [CDI+97] and Torp et al. [TJS04]. We vary the temporal predicate as well as the location of ongoing time intervals to evaluate their effects on runtime and result size.

2.9.1 Setup

The empirical evaluation is conducted on a 3.40 GHz machine with 16GB main memory and an SSD. The client and the database server run on the same machine. We use the PostgreSQL 9.4.0 kernel extended with our implementation of ongoing data types and the operations on them.

Table 2.4 summarizes the real-world and synthetic data sets. As ongoing time intervals we use expanding time intervals $[a, now)$ and shrinking time intervals $[now, b)$. Note that the duration of expanding ongoing time intervals increases as the reference time increases. The earlier an expanding time interval starts, the more time intervals it overlaps with. We use the real-world data sets *MozillaBugs* [LPD13] and *Incumbent* [GSSY98]. The *MozillaBugs* data set records the history of bugs in the Mozilla project. It contains the following three relations. (1) **BugInfo** records general information about a bug: ID, product, component, operating system, textual description, and valid time. Bugs that have not been resolved as of the date of the data export

have ongoing valid time intervals. (2) **BugAssignment** records the email address of the person assigned to a bug, the bug id, and the valid time. (3) **BugSeverity** records the bug id, the severity of the bug, and the valid time. The last assignment and last severity of bugs with ongoing valid times have ongoing valid times as well. *Incumbent* records the valid time periods during which projects are assigned to university employees. We converted project assignments that were not finished at the date of the data export into tuples with ongoing assignments, resulting in 19% ongoing tuples.

Table 2.4: Characteristics of the experiment data sets.

(a) Real-world data sets.

	MozillaBugs			**Incumbent**
	BugInfo B	**BugAssignment A**	**BugSeverity S**	
Cardinality	394,878	582,668	434,078	83,852
# ongoing	60,372 (15%)	63,588 (11%)	61,113 (14%)	15,805 (19%)
Time intervals	$[a, now)$	$[a, now)$	$[a, now)$	$[a, now)$
Time span	20 years	20 years	20 years	16 years

(b) Synthetic data sets.

	$\mathbf{D^{ex}}$	$\mathbf{D^{sh}}$	$\mathbf{D^{sc}}$
Cardinality	10M	10M	35M
#ongoing	15%	15%	20%
Time intervals	$[a, now)$	$[now, b)$	$[a, now)$
Time span	10 years	10 years	10 years

Figure 2.8 shows the distribution of the start points of the ongoing time intervals. In *MozillaBugs*, 50% of the tuples with ongoing time intervals in relations **BugInfo**, **BugAssignment**, and **BugSeverity** are located within the last two years of the history. In *Incumbent*, all ongoing project assignments started within the last year of the history. For experiments with an increasing number of tuples we grow the size of the real-world data sets by growing the history backward. This means that the percentage of ongoing time intervals decreases as the data size grows. For *MozillaBugs*, we grow the history backward for the **BugInfo** relation and use all records in the other two relations that match to the bug ids in **BugInfo**.

To maximize performance we implemented the bind operator of Clifford et al. [CDI+97] in the PostgreSQL 9.4.0 kernel as a C function that is called when an ongoing attribute is accessed [TJS04]. $\text{Cliff}_{\text{max}}$ refers to Clifford's approach that uses a reference time that is greater than the latest end point. It represents the typical use case with reference times close to the current time.

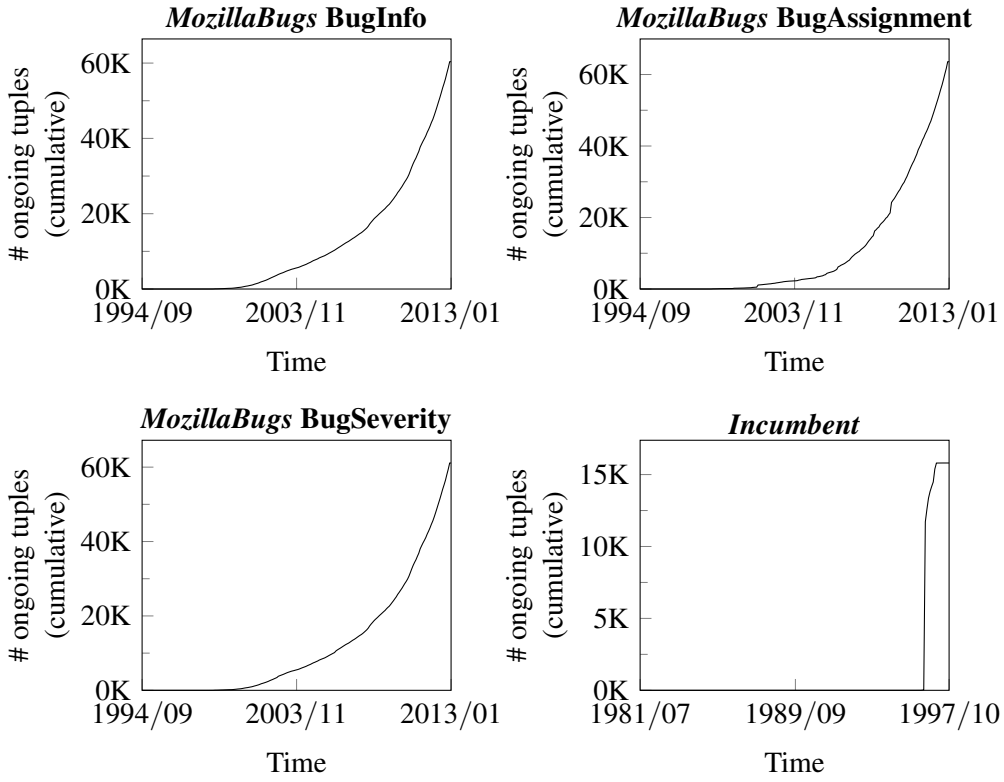


Figure 2.8: Start point distribution of ongoing intervals.

We use two relational algebra operators for the evaluation: selection $Q_i^{\sigma} = \sigma_{VT\ \mathrm{pred}_i\ [t_s,t_e)}(\mathbf{R})$ with a temporal predicate on the valid time and join $Q_i^{\bowtie} = \mathbf{R} \bowtie_{\theta_N \wedge\ \mathbf{R}.VT\ \mathrm{pred}_i\ \mathbf{S}.VT} \mathbf{S}$ whose join predicate includes equality predicates on non-temporal attributes (θ_N) and a temporal predicate pred_i on the valid time. **S** and **R** refer to the same relation. The fixed time interval $[t_s,t_e)$ in the selection predicate spans the last 10% of the data history. Selection is a fast operator and will show the overhead of our approach; join queries are common for database systems and representative for different workloads. On *MozillaBugs*, we use a complex join query to evaluate our approach on a heavier workload as well. The join query determines for a person similar bugs that are open at any time when the person is working on a bug with severity *major*. Similar bugs are bugs that affect the same product, component, and operating system (θ_{sim}):

$$QC_i^{\bowtie} = \mathbf{A} \bowtie_{\mathbf{A}.ID=\mathbf{S}.ID \wedge \mathbf{A}.VT\ \mathrm{overlaps}\ \mathbf{S}.VT \wedge \mathrm{Severity='major'}} \mathbf{S}$$
$$\bowtie_{\mathbf{A}.ID=\mathbf{B}.ID} \mathbf{B} \bowtie_{\theta_{sim} \wedge \mathbf{A}.VT\ \mathrm{pred}_i\ \mathbf{B}'.VT} \mathbf{B}'$$

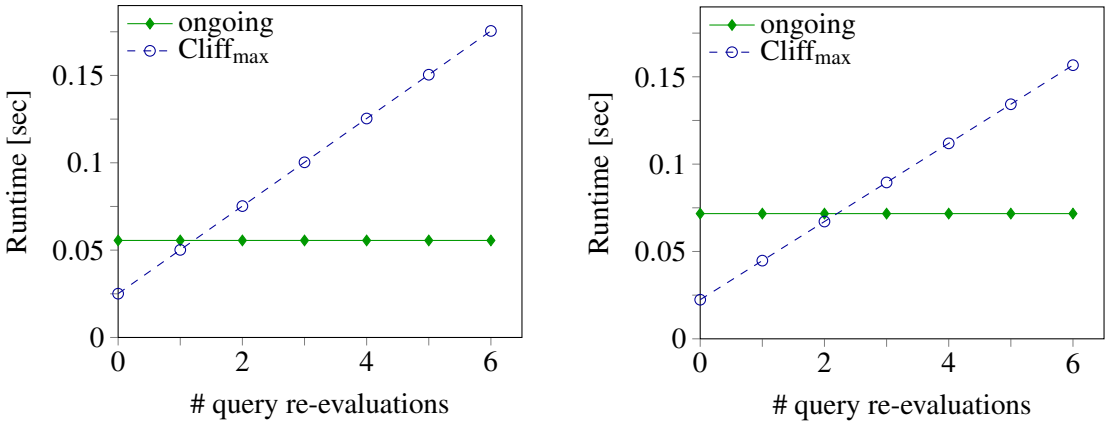
As temporal predicates, we use *overlaps* ($\mathrm{pred}_{\mathrm{ovlp}}$) and *before* ($\mathrm{pred}_{\mathrm{bef}}$). These predicates are representative for the most commonly used temporal predicates [TPC17, DBG14, CB17, BM17,

PHD16]. The ongoing approach uses the predicates for ongoing time intervals (cf. Section 2.6). To maximize the performance of Clifford's approach, we use the predicates for fixed time intervals.

2.9.2 Query Re-Evaluations

Our approach evaluates a query to an *ongoing result* that is returned to an application. Since ongoing results do not get invalidated by time passing by, the application does not have to re-evaluate the query. In contrast, Clifford's query results get invalidated as time passes by and thus, the application must re-evaluate the query. First, we evaluate the break-even point of the ongoing approach for different predicates. Next, we evaluate the impact of the location and number of ongoing time intervals on the runtime.

Number of Query Re-Evaluations The ongoing approach has a runtime overhead due to the handling of the predicates on ongoing time points and time intervals and due to possibly larger result sizes (cf. Section 2.9.4). This is shown in Figure 2.9 on the real world data *Incumbent* for the temporal predicates *overlaps* and *before*. Clearly, the ongoing approach already performs better after very few query re-evaluations. Specifically, the ongoing approach is faster after two re-evaluations for the *overlaps* predicate (Figure 2.9a) and after three re-evaluations for the *before* predicate (Figure 2.9b). Selection Q^{σ}_{ovlp} is faster than selection Q^{σ}_{bef} for ongoing time intervals because the optimized implementation of the *overlaps* predicate requires about half as many fixed-value comparisons per tuple as the *before* predicate.

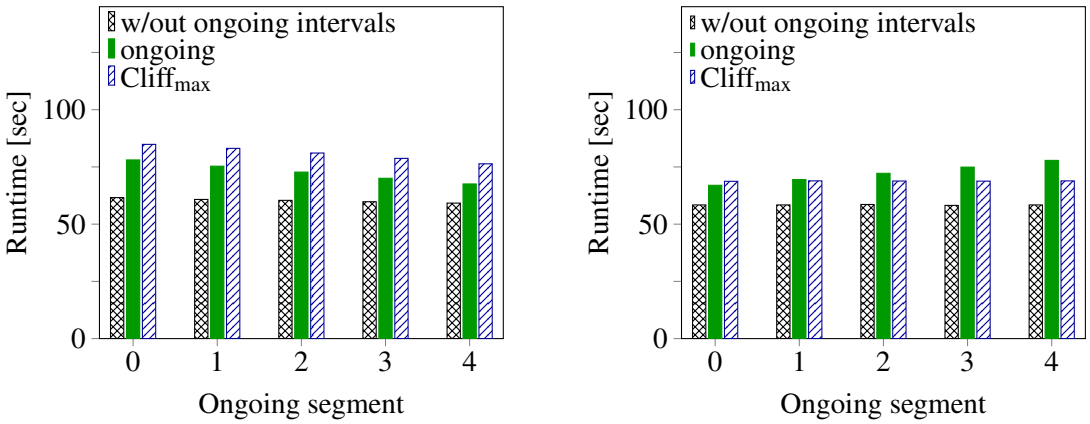


(a) Q^{σ}_{ovlp} with *overlaps*.

(b) Q^{σ}_{bef} with *before*.

Figure 2.9: Number of query re-evaluations on *Incumbent*.

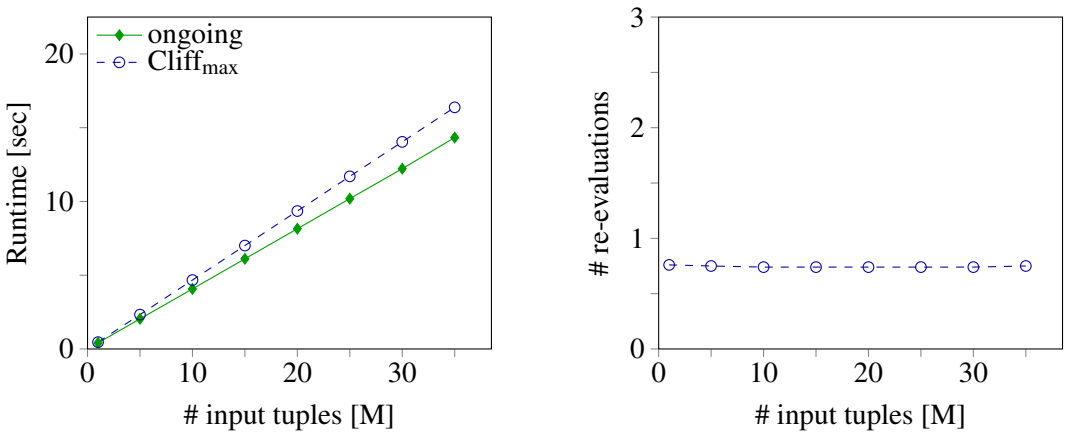
Location of Ongoing Time Intervals We vary the location of the ongoing time intervals by dividing the 10 year history into 5 segments (2 years each) and placing all start points (D^{ex}) or end points (D^{sh}) of the ongoing intervals into one of the segments. Ongoing segment 0 spans the first two years. Figure 2.10 shows the impact of the location on the runtime for one re-evaluation. Since D^{ex} contains expanding ongoing time intervals, the runtime of the ongoing approach decreases for the *overlaps* predicate if the ongoing time intervals are placed in the later segments (cf. Figure 2.10a). Figure 2.10b shows that the opposite observation holds for shrinking ongoing time intervals in D^{sh} since their duration is longer when their end points are placed in later ongoing segments. To establish a baseline for the runtime, we replaced all ongoing time intervals in the two data sets with fixed time intervals and evaluated query $Q^{\bowtie}_{\text{ovlp}}$ on these data sets (without ongoing time intervals). Observe that the baseline runtime accounts for 80% to 90% of the runtime of the ongoing approach. Thus, the join processing is the expensive part and the runtime overhead for processing ongoing time intervals is less than 20%.



(a) $Q^{\bowtie}_{\text{ovlp}}$ on D^{ex}. (b) $Q^{\bowtie}_{\text{ovlp}}$ on D^{sh}.

Figure 2.10: Location of ongoing time intervals.

Number of Input Tuples We evaluate the scalability by increasing the size of the input relation. Figure 2.11a shows that the ongoing approach has a similar linear runtime increase as Clifford's approach does with increasing input sizes. Thus, as shown in Figure 2.11b, the number of query re-evaluations after which the ongoing approach performs better stays constant as the number of input tuples increases.



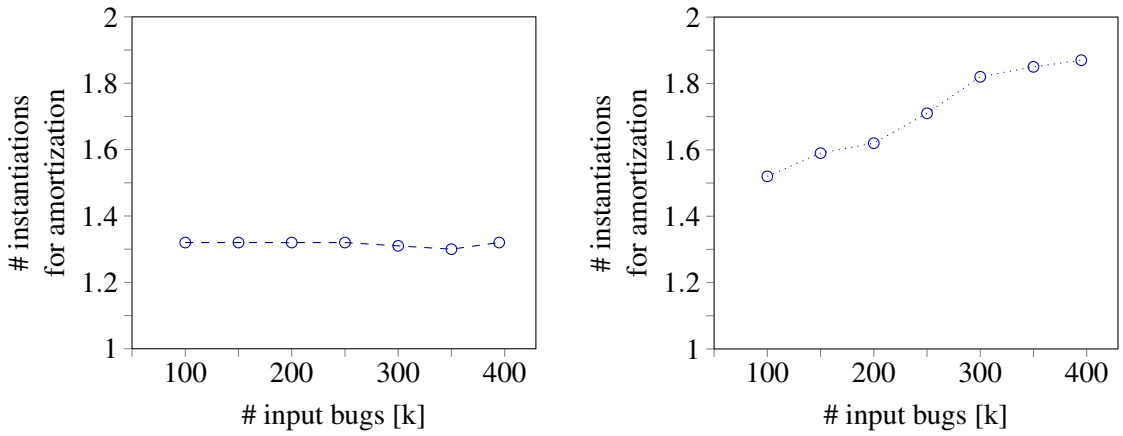
(a) Runtime. (b) Query re-evaluations.

Figure 2.11: Number of input tuples (Q^{σ}_{ovlp} on D^{sc}).

2.9.3 Instantiated Query Results via Materialized Views

Ongoing relations can easily be combined with materialized views to efficiently compute instantiated results at different reference times. This allows applications that do not want to handle ongoing relations explicitly to leverage the performance benefits of ongoing relations. We evaluate the runtime amortization of the ongoing approach, i.e., at how many different reference times n an instantiated result must be returned to an application, such that calculating the ongoing result and instantiating it at the n reference times outperforms Clifford's approach, which must calculate the query at each of the n reference times. The main factors for the amortization are (1) the complexity of the query and (2) the reference time used for the instantiation.

Query Complexity Figure 2.12 shows the amortization for selection and complex join. The number of input bugs (x-axis) is equal to the number of tuples in relation **B** (cf. Section 2.9.1 on how we vary the size of the data set). Both queries require less than two instantiations for the amortization at all input sizes. For the selection query, the number of reference times for amortization remains constant with varying input size. For the complex join, it increases slightly: around 25% for a 300% input bugs increase. This is because the query optimizer chooses a linear-time hash join for Clifford's approach when evaluating the join with $\mathbf{B}'$, whereas it uses a log-linear-time merge join for the ongoing approach. This additional logarithmic component is consistent with the curve in Figure 2.12b.



(a) Selection $Q^{\sigma}_{\text{ovlp}}(\mathbf{B})$. (b) Join $QC^{\bowtie}_{\text{ovlp}}(\mathbf{A},\mathbf{S},\mathbf{B})$.

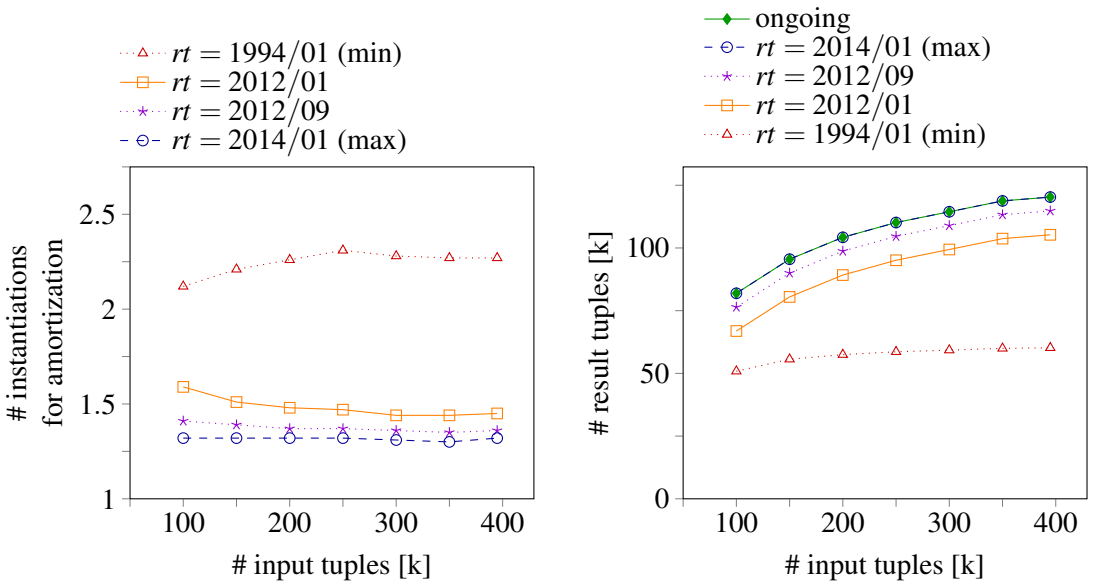
Figure 2.12: Amortization for selection and join on *MozillaBugs*.

Reference Time Smaller size differences of the ongoing and instantiated query result lead to a faster runtime amortization of the ongoing approach. The size of the ongoing result is independent of the reference time whereas the size of the instantiated result depends on it. Figure 2.13a shows that the amortization of the ongoing approach decreases from three instantiations for early reference times ($rt = \text{min}$, i.e., smallest time point in the data set) to two instantiations for later reference times. For the *overlaps* predicate, later reference times result in smaller size differences: the later the reference time, the more ongoing time intervals instantiate to non-empty time intervals. Thus, more and more ongoing time intervals satisfy the predicate (especially as a late selection time interval is used) and belong to the result (Figure 2.13b).

2.9.4 Storage

The ongoing approach requires additional storage for each tuple and for the tuples that belong to the ongoing result but not to Clifford's result. The per-tuple storage overhead is the additional *RT* attribute and a doubling of the size of the valid time attribute (because ongoing rather than fixed values are used). Typically, the value of the *RT* attribute can be represented with one fixed time interval.

Per-Tuple Storage The per-tuple storage overhead consists of the additional *RT* attribute and doubling the size of the valid time (+8 Bytes). We first analyze the cardinality of the *RT* attribute,



(a) Amortization. (b) Result size.

Figure 2.13: Amortization for $Q^{\sigma}_{\text{ovlp}}(\mathbf{B})$ on *MozillaBugs*.

i.e., the number of fixed intervals that is needed to represent a tuple's reference time, and then discuss the additional storage requirements.

Table 2.5 shows that the result of the common predicates on ongoing time intervals (cf. Table 2.3) can be represented with one interval in most cases.

Table 2.5: Predicates: maximum cardinality of RT.

	Ongoing time intervals		
	expanding	shrinking	expanding + shrinking
before	1	1	1
starts	1	1	1
during	1	1	1
meets	1	1	1
finishes	1	1	1
equals	1	1	1
overlaps	1	1	2

Thus, the typical input cardinality for subsequent logical connectives is one. For conjunction $\mathbf{b}_1 \wedge \mathbf{b}_2$ and disjunction $\mathbf{b}_1 \vee \mathbf{b}_2$ the worst case output cardinality is $|\mathbf{b}_1| + |\mathbf{b}_2|$. Negation has an output cardinality of $|\mathbf{b}_1| - 1 \leq |\neg \mathbf{b}_1| \leq |\mathbf{b}_1| + 1$. Conjunction is the most widely used connective in predicates and is used to restrict a tuple's reference time. Its typical output cardinality is one. Thus, the typical cardinality of *RT* is one as well.

Table 2.6 shows the per-tuple storage requirements for the three base relations of the *MozillaBugs* data set and two query results. The *RT* attribute contributes 29 Bytes to the storage size of a tuple in all five relations. This corresponds to the typical case where a tuple's reference time is represented with one fixed time interval. The constant overhead for the *RT* attribute can be significant for small tuple sizes (+75% for 100B) and gets insignificant for larger tuples (+4% for $\geq$ 1kB). Small tuple sizes often occur in foreign key relations. Larger tuple sizes occur in real-world data with descriptive attributes (e.g., the textual description of a bug).

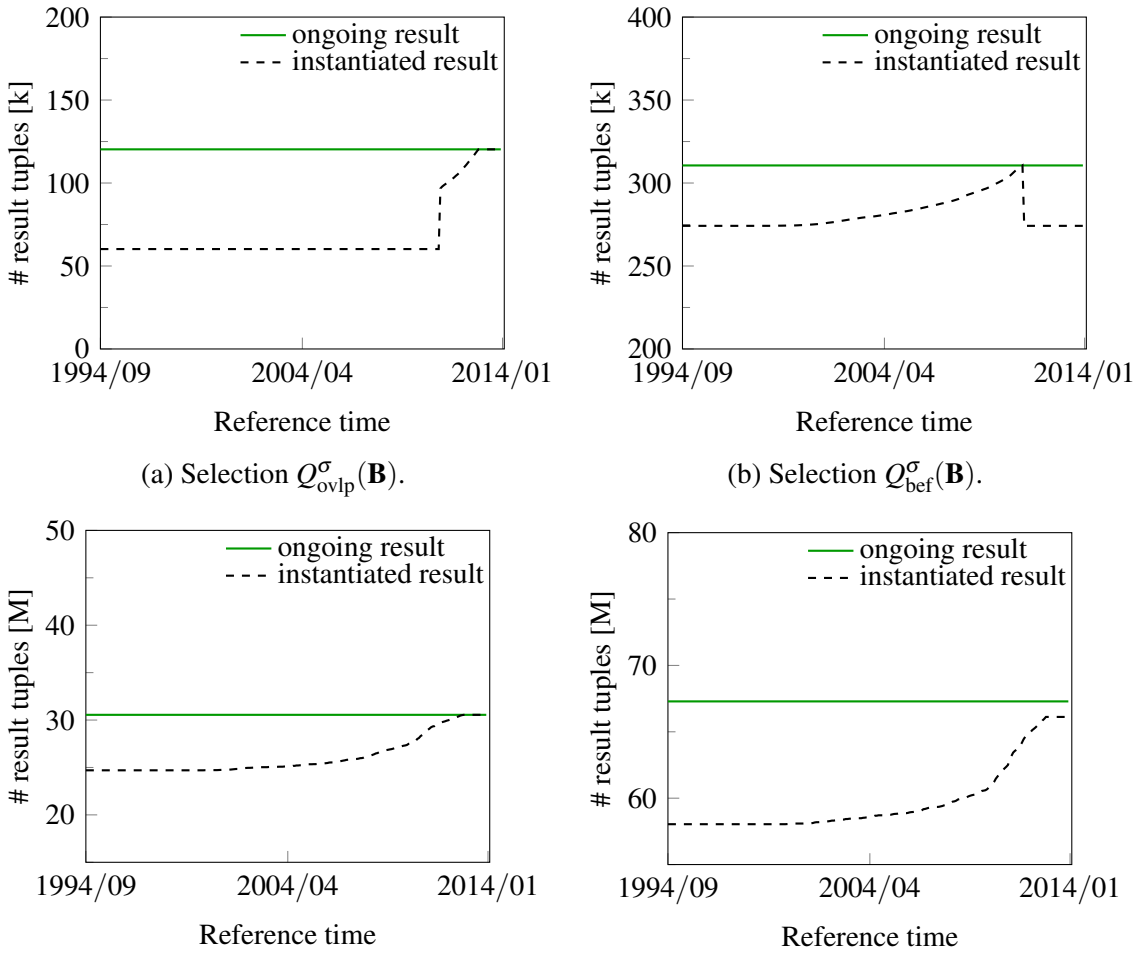
Table 2.6: Per-tuple storage on *MozillaBugs*.

	B	**A**	**S**	$Q^{\sigma}_{\text{ovlp}}(\mathbf{B})$	$QC^{\bowtie}_{\text{ovlp}}$
avg tuple size	968B	90B	86B	968B	2.34kB
RT size	29B (3%)	29B (32%)	29B (34%)	29B (3%)	29B (1%)
$\frac{\text{ongoing}}{\text{fixed}}$ tuple size	104%	167%	175%	104%	103%

The number of additional tuples that are part of the ongoing result but not of Clifford's result depends on the reference time. Since ongoing results combine the results at all reference times, they must contain at least the tuples of the largest instantiated result. If the size of the ongoing result and the largest instantiated result are equal, the size of the ongoing result is optimal.

For expanding ongoing intervals the size of the ongoing result is optimal for predicate *overlaps* (Figure 2.14a and Figure 2.14c). As the duration of expanding time intervals increases, once an expanding time interval overlaps with a time interval, they remain overlapping for all reference times afterwards. Tuples are only added to the instantiated query results with increasing reference times and thus, the ongoing result contains exactly the tuples of the largest instantiated result.

For expanding ongoing intervals and the *before* predicate, the ongoing result reaches the optimal size for selections (Figure 2.14b) and gets close to it for joins (Figure 2.14d). Due to the duration increase, expanding ongoing time intervals are before a time interval up to a reference time and then stop being before it. As there is one selection interval in the selection, this reference time is the same for all expanding time intervals (it is the start point of the selection interval). In a join, an expanding time interval is compared to multiple time intervals. Usually there does not exist a single reference time that belongs to the *RT* attribute of all result tuples, and thus, the maximum instantiated result is smaller than the ongoing result.



(a) Selection $Q^{\sigma}_{\text{ovlp}}(\mathbf{B})$.

(b) Selection $Q^{\sigma}_{\text{bef}}(\mathbf{B})$.

(c) Join $QC^{\bowtie}_{\text{ovlp}}(\mathbf{A},\mathbf{S},\mathbf{B})$.

(d) Join $QC^{\bowtie}_{\text{bef}}(\mathbf{A},\mathbf{S},\mathbf{B})$.

Figure 2.14: Result size vs. reference time on *MozillaBugs*.

2.9.5 Summary

As expected, the ongoing approach has a runtime overhead to compute ongoing results that do not get invalidated by time passing by. This overhead is quite small and pays off for as little as three query re-evaluations of Clifford's approach when returning an ongoing result and for returning as little as two instantiated results when leveraging the ongoing result to calculate them. For late reference times, which are close to the current time, the result size of the ongoing approach is equal to the result size of Clifford's approach for the widely-used *overlaps* predicate and close to equal for other predicates. Thus, the number of tuples that are contained in an ongoing result but not in Clifford's result is small.

2.10 Conclusions

We propose the first approach that evaluates queries on ongoing relations without instantiating ongoing time points. Ongoing time points are preserved in query results and the results remain valid as time passes by. For database systems this is a crucial property as it guarantees that cached results, materialized views, etc. have to be maintained only after explicit database modifications. We define predicates and functions on ongoing time points and time intervals. We propose *ongoing relations* that associate each tuple with a reference time attribute. The value of the reference time attribute contains the reference times when a tuple belongs to the instantiated relations and is restricted by predicates on ongoing attributes.

There are several interesting topics for future research. First, we want to extend the set of functions for ongoing data types to include a duration function for ongoing time intervals whose result are ongoing integers. Second, we plan to propose an aggregation operator for ongoing relations and determine the additional ongoing data types that are required to support aggregation and group tuples in the presence of RT and ongoing attributes. Finally, we want to develop index access methods for ongoing time points (based on the approaches for indexing fixed time intervals) and discuss query classes that benefit from these indexes.

CHAPTER 3

Functions on Ongoing Intervals

Abstract

Intersection, difference, and union are standard functions and building blocks for processing time intervals. The presence of the ongoing time point *now* in the data significantly complicates the logic of these functions. To shield applications from this complexity it is important that interval functions transparently handle ongoing intervals, such as $[08/17, now)$. In particular, interval functions must evaluate to intervals that are correct for all possible values of *now* and they must correctly group time points into intervals so that subsequent predicates, e.g., *during*, evaluate to the correct truth values.

This paper proposes the first solution that evaluates interval functions to the expected time intervals at each reference time, i.e., at each possible value of *now*. We propose function results that are pairs consisting of an ongoing interval and the reference times when this ongoing interval is part of the result. For the representation we leverage ongoing relations with a single reference time attribute that integrates the restrictions from the results of all interval functions. We describe and evaluate an efficient implementation of the interval functions in PostgreSQL.

3.1 Introduction

Temporal relational algebra operators [DBGJ16, BJ09] and temporal modifications [TJS04, KM12] have to adjust the valid time of the input tuples. To do the adjustment, they apply *sequences of interval functions* to the original valid times. For time intervals the intersection, difference, and union function are used to do the adjustment. A representative example is the temporal anti-join $\mathbf{R} \triangleright^{\mathrm{T}}_{\theta} \mathbf{S}$. It adjusts the valid time of each tuple $r \in \mathbf{R}$ by subtracting the valid times of all tuples $s \in \mathbf{S}$ that satisfy predicate θ. Thus, the temporal anti-join uses a sequence of difference functions on time intervals to adjust valid times.

We provide a solution for defining and implementing functions on ongoing intervals that include time point *now*. Since ongoing data types change their value as time passes by, the result of functions on ongoing data types must also change as time passes by.

The reference time determines the value of ongoing time point *now*. Our goal is to evaluate functions on ongoing intervals to results that remain valid. This means that, at each reference time, the function result must be equal to the time intervals one would expect if all occurrences of *now* had been replaced by the reference time. For example, for the difference function $T_1 - T_2$ the expected time intervals are the maximal sub-intervals of T_1 that do not overlap with T_2. Formally, at each possible reference time rt, the result of a function f on ongoing intervals T_1 and T_2 is equal to the result obtained by evaluating the corresponding function f^F for fixed time intervals on the instantiated time intervals:

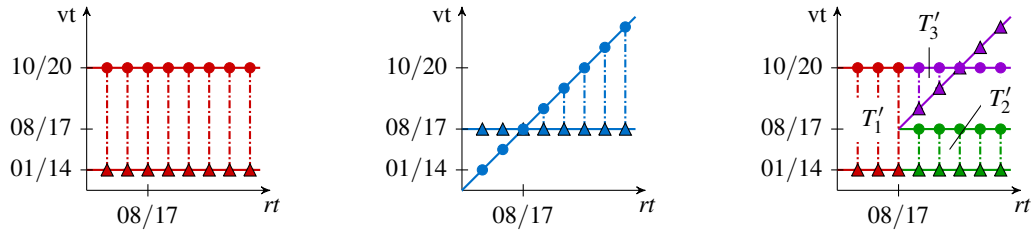
$$\forall rt(\|f(T_1,T_2)\|_{rt} = f^F(\|T_1\|_{rt},\|T_1\|_{rt})$$

Fixed time intervals are time intervals without ongoing start and end points. The bind operator $\|\cdot\|_{rt}$ replaces all occurrences of *now* with the reference time.

Example 12. *Consider the difference* $[01/14, 10/20) - [08/17, now)$. *The input time intervals and the difference result are illustrated in Figure 3.1.*

The result in Figure 3.1c consists of the following pairs of ongoing interval and reference times rt:

- $T_1' = [01/14, 10/20)$ *at* $rt \in (-\infty, 08/18)$,
- $T_2' = [01/14, 08/17)$ *at* $rt \in [08/18, \infty)$, *and*
- $T_3' = [now, 10/20)$ *at* $rt \in [08/18, \infty)$.



(a) $T_1 = [01/14, 10/20)$ (b) $T_2 = [08/17, now)$ (c) $T_1 - T_2$

Figure 3.1: The difference result with the expected time intervals at each reference time.

Up to reference time $08/17$*, time interval* $[08/17, now)$ *instantiates to empty time intervals (cf. Figure 3.1b) and the input time intervals do not overlap. Thus, the difference result is the input interval* $[01/14, 10/20)$ *for reference times in* $(-\infty, 08/18)$*. From reference time* $08/18$ *on, time interval* $[08/17, now)$ *instantiates to non-empty time intervals and the input time intervals overlap. The difference result are the sub-intervals* $[01/14, 08/17)$ *and* $[now, 10/20)$ *for reference times in* $[08/18, \infty)$.

Evaluating functions to the expected time interval at each reference time is necessary (1) to get the correct truth value for predicates (e.g., *during*) over the result time intervals and (2) for temporal relational algebra operators that require the expected intervals for their correctness [DBGJ16, DGN+19].

Our solution supports the intersection, difference, and union function on ongoing intervals. We evaluate these functions to results that remain valid as time passes by and represent their results as a combination of ongoing intervals and the reference times when the ongoing intervals are part of the result. Functions return sets of (ongoing interval, RT)-pairs as illustrated in Figure 3.1.

Our implementation uses ongoing relations [MB20b] to store the results of interval functions in databases. Ongoing relations associate each tuple with a reference time attribute *RT* whose value are the reference times when the tuple belongs to the relation. An ongoing relation stores a function result as follows: the ongoing interval is stored in its own attribute and the reference time of the tuple is restricted to the original reference time of the tuple intersected with the reference times of the ongoing interval. For set-valued function results, the result relation contains a tuple for each value.

Relational algebra operators with multiple and nested functions are flattened into nested projections, so that each projection list includes one interval function at most. For multiple functions the flattening is equivalent to the cross product of the result intervals. For nested functions it is

equivalent to applying the enclosing function to each result interval of nested functions. A single reference time attribute is sufficient to incorporate the reference time restrictions of all interval functions.

We have extended the PostgreSQL kernel to support the intersection, the difference, and the union function for ongoing intervals. For this we have implement the functions as set-returning functions [Pos20c] that return a set of (ongoing interval, RT)-pairs, one pair at each invocation. Using these pairs as return values minimizes the number of result pairs and keeps the size of the result relation minimal.

Our contributions are the following:

- We define the intersection, difference and union function for ongoing intervals, such that their results remain valid. We represent their results as (ongoing interval, RT)-pairs.

- We use ongoing relations to store the results of interval functions. We store the ongoing interval in its own attribute and set the result tuple's reference time to the reference times when the tuple and the ongoing value are part of the result.

- We evaluate nested and multiple functions on ongoing intervals in relational algebra operators with nested projections for each function. This guarantees the correct restriction of the result tuple's reference time with set-valued functions.

- We provide an efficient implementation of the intersection, difference, and union function on ongoing intervals in the kernel of PostgreSQL. We implement these functions as set-returning functions that minimize the number of result tuples.

The paper is organized as follows. Section 3.2 introduces our running example. Section 3.3 discusses related work. Section 3.4 provides preliminaries. Section 3.5 discusses our choice of representing the function results as a combination of ongoing intervals and the reference time and defines the three standard functions on ongoing time intervals, such that their results remain valid. Section 3.6 discusses the evaluation of relational algebra operators with multiple and nested functions. Section 3.7 discusses the implementation of our solution in PostgreSQL and Section 3.8 describes its evaluation. Section 3.9 concludes the paper and points to future research.

3.2 Running Example

Consider a consulting company that runs software development projects for its clients. An employee has a fixed-term or a permanent employment. Fixed-term employments have fixed start points that indicate the start of the employment and fixed end dates that indicate the end of the employment. Permanent employments have fixed start dates but end dates that keep increasing until the contract is modified. These end dates are *ongoing*. Employees get assigned to software development projects. The projects have different priorities and employees that are assigned to high-priority projects (priority equal to five) cannot be re-assigned to other projects until these projects have been completed. Selected relations of our running example are shown in Figure 3.2 and discussed below.

E

	Name	Role	VT	RT
e_1	Ann	SWE	$[01/14, 10/20)$	$\{(-\infty, \infty)\}$
e_2	Bob	SWE	$[06/15, now)$	$\{(-\infty, \infty)\}$

P

	PID	Priority	Name	VT	RT
p_1	500	5	Ann	$[01/14, 06/15)$	$\{(-\infty, \infty)\}$
p_2	501	5	Ann	$[08/17, +10/20)$	$\{(-\infty, \infty)\}$
p_3	501	5	Bob	$[10/20, +11/22)$	$\{(-\infty, \infty)\}$

Figure 3.2: Relations with ongoing time points.

Relation **E** lists selected employees. An employee is described by her name, her role, the valid time VT during which she is employed at the company, and the reference time RT. Tuple e_1 records employee Ann who has a fixed-term employment and is employed as software engineer (SWE) from $01/14$ until $10/20$; the tuple's ongoing values can be instantiated at all reference times, i.e., it is part of the result at all reference times. Tuple e_2 records employee Bob who has a permanent employment and works as software engineer from $06/15$ onward.

Relation **P** illustrates the assignments of projects to employees. An assignment is described by the project ID (*PID*), the priority of the project, the name of the employee, the valid time interval VT during which the employee is assigned to the project, and the reference time RT. For instance, tuple p_2 records that employee Ann is assigned to a high-priority project with ID 501. The assignment is valid from $08/17$ until no later than $10/20$ and the tuple's ongoing values can be instantiated at all reference times.

To fill the available head-count for their projects, project managers look for employees who are available for at least a project-specific time interval. Thus, the managers are interested in the

time intervals during which a software engineer can be assigned to their projects. The following query determines this information for a project-specific time interval $[07/16, 09/18)$:

$$\mathbf{H} \leftarrow \sigma_{[07/16,09/18) \text{ during VT}} \big(\mathbf{E} \rhd^{\mathrm{T}}_{\mathbf{E}.\mathrm{Name}=\mathbf{P}.\mathrm{Name}} (\sigma_{\mathrm{Priority}=5}(\mathbf{P})) \big)$$

Figure 3.3 illustrates the result of this query. The temporal anti-join adjusts the valid time of the input tuples for the result tuples at each reference time. The valid time of the input tuples is adjusted to the valid times when a software engineer is employed but does not work on a high-priority project. Algorithm 2 calculates the temporal anti-join $\mathbf{E} \rhd^{\mathrm{T}}_{\mathbf{E}.\mathrm{Name}=\mathbf{P}.\mathrm{Name}} (\sigma_{\mathrm{Priority}=5}(\mathbf{P}))$. For each tuple $e \in \mathbf{E}$, each tuple $p \in \sigma_{\mathrm{Priority}=5}(\mathbf{P})$ is used to adjust the valid time of tuple e. Thus, if predicate $\theta(e,p) = (e.\mathrm{Name} = p.\mathrm{Name})$ is satisfied, the valid time of p is subtracted from the valid time of tuple e at their common reference times. If predicate $\theta(e,p)$ is not satisfied, the valid time of tuple e is kept as is (line 12). Observe that Algorithm 2 uses a projection with the difference function to adjust the valid time (line 8). It is transparent to the algorithm that the difference is applied to ongoing valid times. The result of the difference function consists of three (ongoing interval, RT)-pairs (cf. Figure 3.1). The projection decomposes each result pair: it stores the ongoing interval in its own attribute, here VT, and it sets the RT value of the result tuple to the reference times when tuple x belongs to the relation and the ongoing interval is part of the difference result (cf. Section 3.6 for the details). The projection with the difference function returns a tuple for each result pair. For instance, the projection with the difference $e_2.VT - p_3.VT$ results in the three result tuples a_4, a_5, and a_6 in Figure 3.3. Observe that the adjusted valid times in one iteration are used as input for the adjustment in the next iteration (lines 4, 6, 8, 13). For instance, the valid time of input tuple e_1 is adjusted with the tuples p_1 and p_2, which satisfy predicate $\theta(e,p) = (\mathbf{E}.\mathrm{Name} = \mathbf{P}.\mathrm{Name})$. The adjusted valid time is the result of the nested difference $((e_1.VT - p_1.VT) - p_2.VT)$.

As illustrated in Figure 3.3, the result of the difference function is a combination of ongoing intervals and the reference times when the ongoing interval is part of the result. For input tuple e_2, the result tuples are a_4, a_5, and a_6. The valid times of these tuples together with their reference time RT are the result of the difference $(e_2.VT - p_3.VT)$. The valid time of tuple a_4 is the difference result at the reference times when $e_2.VT$ and $p_2.VT$ do not overlap: $a_4.RT = \{(-\infty, 10/21)\}$. The valid times of tuples a_5 and a_6 are the difference result at the reference times when $e_2.VT$ and $p_2.VT$ overlap: $a_5.RT = a_6.RT = \{[10/21, -\infty)\}$.

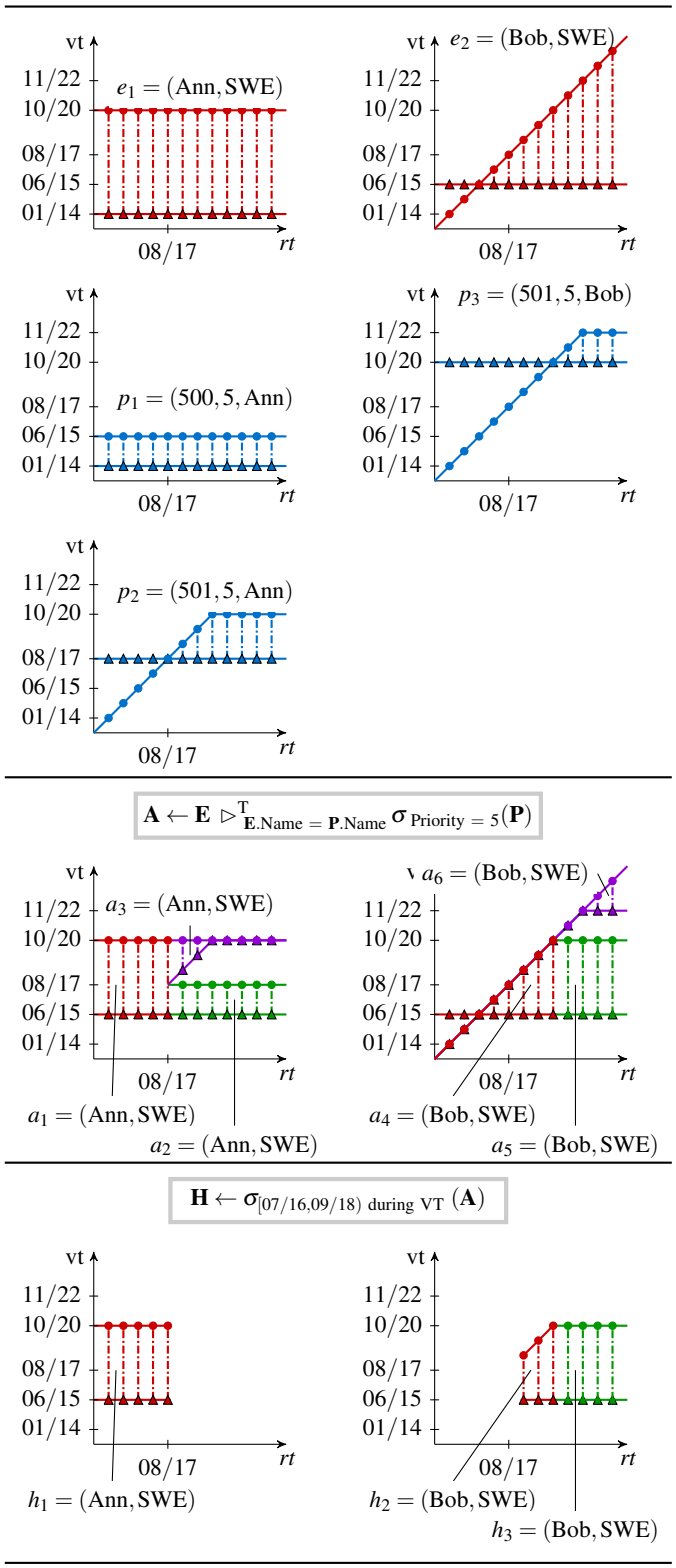


Figure 3.3: The query results remain valid as time passes by.

```
1  R ← {};
2  foreach tuple e ∈ E do
3  |  cR ← {e};
4  |  foreach p ∈ σ_{Priority=5}(P) do
5  |  |  uR ← {};
6  |  |  foreach tuple x ∈ cR do
7  |  |  |  if θ(x, p) then
8  |  |  |  |  aVT ← π_{x.Name, x.Role, (x.VT − p.VT)/VT}(x ∘ p);
9  |  |  |  |  x.RT ← (x.RT ∧ ¬p.RT);
10 |  |  |  |  uR ← uR ∪ aVT ∪ {x};
11 |  |  |  else
12 |  |  |  |  uR ← uR ∪ {x};
13 |  |  cR ← uR;
14 |  R ← R ∪ cR;
15 return R;
```

Algorithm 2: $\mathbf{E} \triangleright^{\mathrm{T}}_{\mathbf{E}.\mathrm{Name}=\mathbf{P}.\mathrm{Name}} (\sigma_{\mathrm{Priority}=5}(\mathbf{P}))$. We write $\pi_{X/C}$ to rename X to C and $x \circ y$ to concatenate tuples.

Note that at each reference time, the valid times of the result tuples are identical to the time intervals to which the difference function for fixed time intervals evaluates to. This allows the subsequent selection to evaluate the *during* predicate on the valid time to the correct truth values. The result of the selection, ongoing relation **H**, includes the tuples shown in Figure 3.3. The selection on ongoing relations selects a tuple by restricting the tuple's reference time RT to the reference times when the predicate is *true*. Tuples with an empty reference time are discarded. For instance, predicate $([07/16, 09/18)$ during $VT)$ for tuple a_4 evaluates to *true* from reference time $09/19$ on. The reference time of tuple a_4, $a_4.RT = \{(-\infty, 10/21)\}$, is restricted with these reference times in result tuple h_2 to $h_2.RT = \{[09/19, 10/21)\}$.

3.3 Related Work

Current database systems support time intervals as basic data types [Ter20b, Pos20b] together with intersection, difference, and union functions to manipulate time intervals [Ter20a, Pos20a]. Since current systems do not support ongoing data types the functions are limited to fixed time intervals, i.e., time intervals without ongoing values.

Various research approaches propose solutions for adding support for ongoing time points and time intervals to database systems [CDI+97, DJTS09, APT16, TJS00, FM96, JL01, TJB97, STS03,

SST09, Sno87], and extending the support for functions and predicates to ongoing time points and time intervals [ASTS13, APS+16, TJS04, MB20b]. Throughout, we use $\mathscr{T}$ to denote the domain of fixed time points.

Anselma et al. [ASTS13] propose a temporal algebra for relations with ongoing time points. The intersection and difference functions on ongoing intervals are used to adjust the time intervals of the input tuples. Their approach evaluates the functions to results that remain valid when the result can be represented with ongoing intervals of $(\mathscr{T} \cup \{now\}) \times (\mathscr{T} \cup \{now\})$. For instance, $[06/15, now) - [01/14, 10/20) = [10/20, now)$. If this is not possible a result for a specific reference time is returned. For instance, $[01/14, 10/20) - [06/15, now)$ evaluates to $\{[01/14, 06/15)$, $[08/17, 10/20)\}$ at reference time $08/17$. Thus, the approach evaluates functions on ongoing intervals to results that do not remain valid as time passes by.

Torp et al. [TJS04] propose a solution for temporal modifications on databases with ongoing time points. Their goal are modifications that remain valid as time passes by. The modifications are defined with intersection and difference functions for ongoing intervals. To represent function results, Torp et al. propose time domain $\mathscr{T}_f = \mathscr{T} \cup \{\min(a, now) | a \in \mathscr{T}\} \cup \{\max(a, now) | a \in \mathscr{T}\}$ with the minimum of a fixed time point and *now* and the maximum of a fixed time point and *now*. By limiting the input time intervals to a subset of $\mathscr{T}_f \times \mathscr{T}_f$, their intersection function evaluates to results that remain valid. This is not the case for the difference function. Torp et al. represent the difference result with ongoing intervals that split the resulting time interval at a reference time into several time intervals. This yields the correct result time points but not the correct grouping of these time points into time intervals. For instance, the result of the difference $(e_1.VT - p_1.VT) - p_2.VT$ in our running example in Section 3.2 is represented with the two ongoing intervals $[06/15, 08/17)$ and $[\max(08/17, now), 10/20)$. At reference time $07/16$, input time interval $(e_1.VT - p_1.VT) = [06/15, 10/20)$ does not overlap with $p_2.VT$ and the difference evaluates to $[06/15, 10/20)$. Torp et al. split this result interval into the intervals $[06/15, 08/17)$ and $[08/17, 10/20)$ in their result. This leads to an incorrect query result since predicate $[07/16$, $09/18)$ during VT in our example query incorrectly evaluates to *false* at reference time $07/16$. Summarizing, their approach can evaluate the intersection function to results that remain valid, but cannot evaluate the difference and union function to such results.

Mülle et al. [MB20b] propose a framework for evaluating predicates and relational algebra operators on ongoing data types to results that remain valid as time passes by. The key idea is to evaluate the operations at every reference time and represent the results as ongoing data types. The result of relational algebra operators are ongoing relations. Their tuples are associated with

a reference time attribute *RT* that restricts the reference times when the tuple can be instantiated. Their approach focuses on relational algebra operators with predicates. Predicates select tuples by restricting a tuple's reference time to the reference times when the predicate is *true*. Mülle et al. propose time domain $\Omega = \{a{+}b | \exists a,b \in \mathscr{T}(a \leq b)\}$. The ongoing time point $a{+}b$ is equal to time point a up to reference time a, equal to the reference time between reference times a and b, and equal to b from reference time b on. Apart from the intersection their approach does not consider functions in relational algebra operators and thus, does not provide a solution to represent the result of the difference and the union function for ongoing intervals, such that their results remain valid.

3.4 Preliminaries

We assume a linearly ordered, discrete time domain $\mathscr{T}$ with $-\infty$ as the lower limit and ∞ as the upper limit. Fixed data types consist of values that do not change as time passes by. Examples include strings, integers, and time points of $\mathscr{T}$. A fixed time interval $[t_s,t_e) \in \mathscr{T} \times \mathscr{T}$ consists of an inclusive fixed start point and an exclusive fixed end point. Ongoing data types consist of values that change as time passes by. Ongoing values can be instantiated to a fixed value with the bind operator $\|\cdot\|_{rt}$ at a reference time rt. Composite ongoing values are instantiated by instantiating each component. Ongoing time point $a{+}b$ instantiates to the following fixed time point at reference time $rt \in \mathscr{T}$:

$$\|a{+}b\|_{rt} = \begin{cases} a & rt \leq a \\ rt & a < rt < b \\ b & \text{otherwise} \end{cases}$$

Ongoing time point $a{+}b$ subsumes fixed time points $a \equiv a{+}a$, time point $now \equiv -\infty{+}\infty$, growing time points $a{+} \equiv a{+}\infty$, and limited time points ${+}b \equiv -\infty{+}b$. An ongoing time interval $[t_s,t_e)$ is a closed-open time interval with ongoing start and end points. At each reference time rt, an ongoing interval instantiates to a fixed time interval by instantiating its start and end point: $\|[t_s,t_e)\|_{rt} = [\|t_s\|_{rt}, \|t_e\|_{rt})$.

$R = (\mathbf{A})$ denotes the schema of a fixed relation $\mathbf{R}$ with fixed attributes $\mathbf{A} = A_1,\ldots,A_n$. A tuple r with schema R is a finite list that contains for every A_i a value from the domain of A_i. A relation $\mathbf{R}$ over schema R is a finite set of tuples over R. The schema of an ongoing relation [MB20b]

consists of fixed and ongoing attributes $A_1, \ldots, A_n$ and the reference time attribute RT: $R = (A_1, \ldots, A_n, RT)$. An ongoing relation $\mathbf{R}$ is instantiated to a fixed relation with the bind operator $\|\mathbf{R}\|_{rt}$ at a reference time rt: $\|\mathbf{R}\|_{rt} = \{(\|r.A_1\|_{rt}, \ldots, \|r.A_n\|_{rt}) \mid r \in \mathbf{R} \wedge rt \in r.RT\}$. $r.A_i$ denotes the value of attribute A_i in tuple r. The value of the reference time attribute RT is a set of reference time points and denotes when the attribute values of a tuple can be instantiated. At all other reference times, the tuple is discarded. The representation of the RT attribute is not relevant for the semantics since it is an internal attribute that cannot be used in functions and predicates. $\theta(r)$ denotes the application of predicate θ to tuple r.

We use the ^{F}-superscript for operations on fixed data types. Operations on fixed data types retain their standard behavior. Thus, $[a,b)$ overlaps^F $[c,d)$ is equal to $(a < d \wedge c < b)$, and $[a, b)$ adjacent^F $[c,d)$ is equal to $(a = d \vee c = b)$. The intersection, difference, and union functions for fixed time intervals are defined as follows:

$$[a,b) \cap^F [c,d) = [\max^F(a,c), \min^F(b,d))$$

$$[a,b) -^F [c,d) = \begin{cases} \{[a,b)\} & \neg([a,b) \text{ overlaps}^F\ [c,d)) \\ \{[a,c), [d,b)\} & \text{otherwise} \end{cases}$$

$$[a,b) \cup^F [c,d) = \begin{cases} \{[\min^F(a,c), \max^F(b,d))\} & [a,b) \text{ overlaps}^F\ [c,d) \\ & \vee\ [a,b) \text{ adjacent}^F\ [c,d) \\ \{[a,b), [c,d)\} & \text{otherwise} \end{cases}$$

These intervals are the *expected intervals*.

Operations on ongoing data types must evaluate to results that remain valid as time passes by. This means that at each reference time rt, the result of an operation op on ongoing values must evaluate to the expected intervals, i.e., the result must be equivalent to the result obtained by first instantiating the ongoing input values $i_1, \ldots, i_n$ at reference time rt and then evaluating the corresponding operation op^F for fixed data types:

$$\forall rt \in \mathscr{T}(\|op(i_1, \ldots, i_n)\|_{rt} \equiv op^F(\|i_1\|_{rt}, \ldots, \|i_n\|_{rt}))$$

Predicates on ongoing values evaluate to different truth values, i.e., *true* or *false*, at different reference times. To account for this we represent the result of a predicate as the reference times when the predicate is *true*. For instance, $[01/14, 10/20)$ overlaps $[08/17, now) = \{[08/18, \infty)\}$,

i.e., the predicate is *true* from reference time $08/18$ onward. The conjunction of two predicates is *true* at the reference times when both predicates are *true*. The disjunction of two predicates is *true* at the reference times when at least one of the predicates is *true*. The negation of a predicate is *true* at the reference times when the predicate is *false*.

3.5 Functions on Ongoing Intervals

3.5.1 (Ongoing Interval, RT)-Result Pairs

Three time domains for ongoing time points have been proposed: $\mathscr{T} \cup \{now\}$ [CDI+97], $\mathscr{T}_f$ [TJS04], and Ω [MB20b]. None of these time domains can represent the result of the difference and union, such that it remains valid as time passes by.

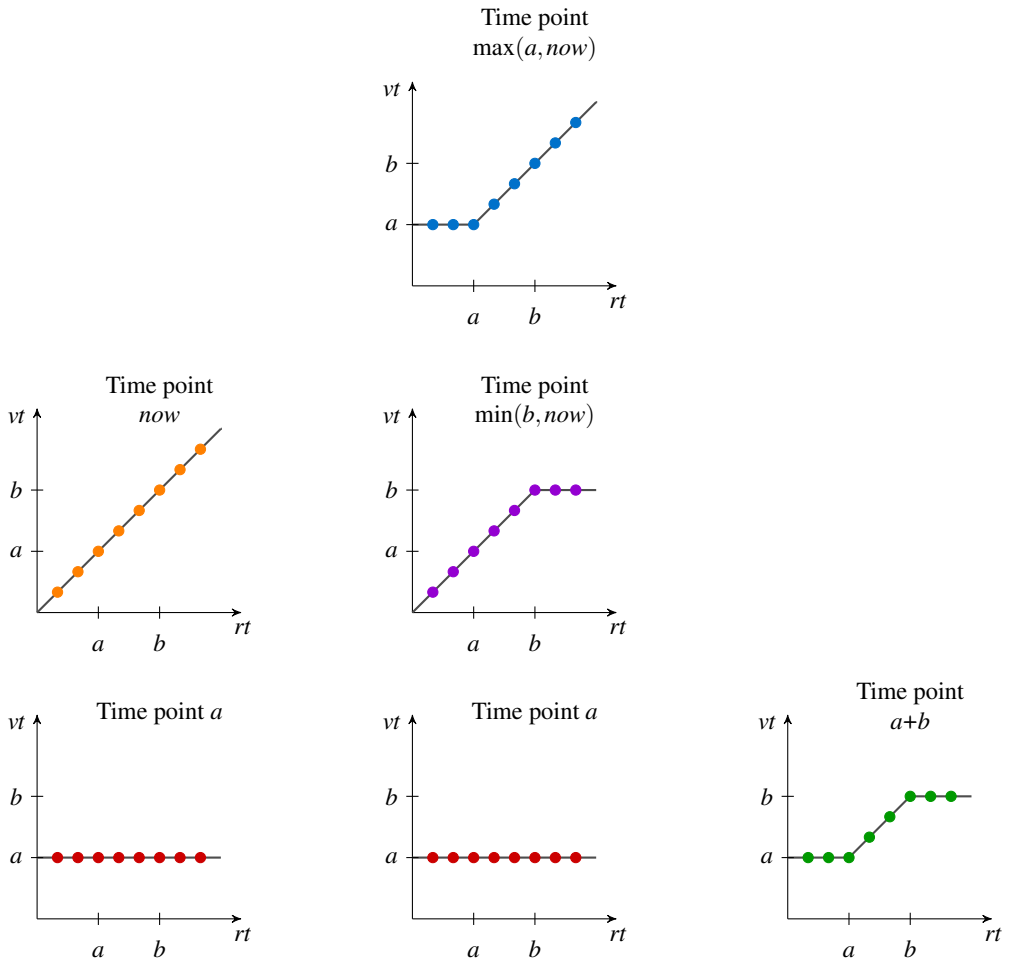
Theorem 1. *Let $\mathscr{D}$ be one of the proposed time domains for ongoing time points. Then, there exist input time intervals, such that the result of the difference $T_1 - T_2$ cannot be represented with time intervals of $\mathscr{D} \times \mathscr{D}$:*

$$\begin{aligned}&\exists T_1, T_2 \in \mathscr{D} \times \mathscr{D}(\\ &\quad \nexists v \in \mathscr{P}(\mathscr{D} \times \mathscr{D})(\forall rt \in \mathscr{T}(\|v\|_{rt} = \|T_1 - T_2\|_{rt})))\end{aligned}$$

Note that the result interval is an element of the power set of all possible time intervals, $\mathscr{P}(\mathscr{D} \times \mathscr{D})$, since the difference function returns several time intervals.

Proof. We prove Theorem 1 with a counterexample. Consider the difference $[01/14, 10/20) - [08/17, now)$. The input time intervals and the result are illustrated in Figure 3.1. At reference time $08/17$, the difference evaluates to time interval $[01/14, 10/20)$ and at reference time $08/18$, it evaluates to time intervals $\{[01/14, 08/17), [08/18, 10/20)\}$. The proposed time domains do not offer ongoing intervals that instantiate to time interval $[01/14, 10/20)$ at reference time $08/17$ and to time intervals $\{[01/14, 08/17), [08/18, 10/20)\}$ at reference time $08/18$. □

Existing time domains are not sufficient to represent the result of interval difference because ongoing time points instantiate to a time point *at each possible* reference time, i.e., $rt \in (-\infty, \infty)$ as illustrated in Figure 3.4.



(a) Domain $\mathcal{T} \cup \{now\}$. (b) Domain $\mathcal{T}_f$. (c) Domain Ω.

Figure 3.4: Illustration of the ongoing time points of the different time domains.

To represent the difference result, we need ongoing time points that can be instantiated *at some* reference times only. We do this by representing the result as a combination of ongoing intervals and the reference time when the ongoing interval is part of the result. The result of the difference $[01/14, 10/20) - [08/17, now)$ can then be represented as

- $[01/14, 10/20)$ at $rt \in (-\infty, 08/18)$,
- $[01/14, 07/18)$ at $rt \in [08/18, \infty)$, and
- $[now, 10/20)$ at $rt \in [08/18, \infty)$.

3.5.2 Definition

Function results that remain valid as time passes by must instantiate to the expected time intervals at each reference time rt. The expected result intervals of function $f(T_1, T_2)$ are maximal time intervals that are equal to the result intervals obtained by evaluating the function on the instantiated input intervals ($T_1, T_2 \in \Omega \times \Omega$):

$$\begin{aligned}\forall rt(\|f(T_1,T_2)\|_{rt} = f(\|T_1\|_{rt}, \|T_1\|_{rt}) \wedge \\ \forall T_x, T_y \in \|f(T_1,T_2)\|_{rt}(\\ T_x \neq T_y \Rightarrow \neg(T_x \text{ overlaps}^F\ T_y) \wedge \neg(T_x \text{ adjacent}^F\ T_y)))\end{aligned}$$

As an example, the difference $[01/14, 10/20) - [08/17, now)$ evaluates to the expected time interval $[01/14, 10/20)$ at reference time $08/17$ and it evaluates to the expected time intervals $[01/14, 08/17)$ and $[08/18, 10/20)$ at reference time $08/18$. These result intervals are maximal and equal to the result obtained by evaluating the difference on the instantiated input time intervals.

Definition 6. *Let T_1, T_2 be two ongoing intervals of $\Omega \times \Omega$. The intersection, difference, and union function are defined as follows:*

Function	***Definition***
$\cap$	$T_1 \cap T_2 = \mathbf{V} \Leftrightarrow \forall rt \in \mathscr{T}(\|\mathbf{V}\|_{rt} = \|T_1\|_{rt} \cap^F \|T_2\|_{rt})$
$-$	$T_1 - T_2 = \mathbf{V} \Leftrightarrow \forall rt \in \mathscr{T}(\|\mathbf{V}\|_{rt} = \|T_1\|_{rt} -^F \|T_2\|_{rt})$
$\cup$	$T_1 \cup T_2 = \mathbf{V} \Leftrightarrow \forall rt \in \mathscr{T}(\|\mathbf{V}\|_{rt} = \|T_1\|_{rt} \cup^F \|T_2\|_{rt})$

We provide concrete values $\mathbf{V}$ for the intersection, the difference, and the union function in Theorem 2.

As an example, the difference on ongoing intervals evaluates to a result that at each reference time is equal to the result obtained by the difference function for fixed time intervals.

Theorem 2. *The results of the functions on ongoing intervals in Definition 6 satisfy the equivalences in Table 3.1.*

Table 3.1: Equivalences for functions on ongoing intervals.

Function	Equivalence
$\cap$	$[t_s,t_e) \cap [\tilde{t}_s,\tilde{t}_e)$
	$\equiv \{([\max(t_s,\tilde{t}_s), \min(t_e,\tilde{t}_e)), \{(-\infty,\infty)\})\}$
$-$	$[t_s,t_e) - [\tilde{t}_s,\tilde{t}_e)$
	$\equiv \{([t_s,t_e), \neg([t_s,t_e)$ overlaps $[\tilde{t}_s,\tilde{t}_e))),$
	$([t_s,\tilde{t}_s), [t_s,t_e)$ overlaps $[\tilde{t}_s,\tilde{t}_e)),$
	$([\tilde{t}_e,t_e), [t_s,t_e)$ overlaps $[\tilde{t}_s,\tilde{t}_e))\}$
$\cup$	$[t_s,t_e) \cup [\tilde{t}_s,\tilde{t}_e)$
	$\equiv \{([t_s,t_e), \neg([t_s,t_e)$ overlaps $[\tilde{t}_s,\tilde{t}_e)) \wedge \neg([t_s,t_e)$ adjacent $[\tilde{t}_s,\tilde{t}_e))),$
	$([\tilde{t}_s,\tilde{t}_e), \neg([t_s,t_e)$ overlaps $[\tilde{t}_s,\tilde{t}_e)) \wedge \neg([t_s,t_e)$ adjacent $[\tilde{t}_s,\tilde{t}_e))),$
	$([\min(t_s,\tilde{t}_s), \max(t_e,\tilde{t}_e)), [t_s,t_e)$ overlaps $[\tilde{t}_s,\tilde{t}_e) \vee [t_s,t_e)$ adjacent $[\tilde{t}_s,\tilde{t}_e))\}$

Proof. We prove the equivalence for the difference function with the result

$$\begin{aligned}\mathbf{V} \equiv \{&([t_s,t_e), \neg([t_s,t_e) \text{ overlaps } [\tilde{t}_s,\tilde{t}_e))),\\ &([t_s,\tilde{t}_s), [t_s,t_e) \text{ overlaps } [\tilde{t}_s,\tilde{t}_e)),\\ &([\tilde{t}_e,t_e), [t_s,t_e) \text{ overlaps } [\tilde{t}_s,\tilde{t}_e))\}\end{aligned}$$

In the proof we use the following properties. The overlaps predicate for ongoing intervals and the negation remain valid as time passes by [MB20b]. The set $\{(v_1,RT_1),\dots\}$ is instantiated at reference time rt by instantiating the value v_i of all the pairs for which rt is contained in RT_i: $\|\{(v_1,RT_1),\dots\}\|_{rt} = \{\|v\|_{rt} | (v,RT) \in \{(v_1,RT_1),\dots,\} \wedge rt \in RT\}$.

At a reference time rt, the input time intervals either overlap or do not overlap. We show the correctness for both cases.

Case 1: $\|[t_s,t_e)\|_{rt}$ overlaps^F $\|[\tilde{t}_s,\tilde{t}_e)\|_{rt}$ is *true*.

$$\begin{aligned}\|\mathbf{V}\|_{rt} &= \|[t_s,t_e)\|_{rt} -^F \|[\tilde{t}_s,\tilde{t}_e)\|_{rt}\\ \{\|[t_s,\tilde{t}_s)\|_{rt}, \|[\tilde{t}_e,t_e)\|_{rt}\} &= \{[\|t_s\|_{rt}, \|\tilde{t}_s\|_{rt}), [\|\tilde{t}_e\|_{rt}, \|t_e\|_{rt})\}\\ \{\|[t_s,\tilde{t}_s)\|_{rt}, \|[\tilde{t}_e,t_e)\|_{rt}\} &= \{\|[t_s,\tilde{t}_s)\|_{rt}, \|[\tilde{t}_e,t_e)\|_{rt}\}\end{aligned}$$

Case 2: $\neg^F(\|[t_s,t_e)\|_{rt}$ overlaps^F $\|[\tilde{t}_s,\tilde{t}_e)\|_{rt})$ is *true*.

$$\|\mathbf{V}\|_{rt} = \|[t_s,t_e)\|_{rt} -^F \|[\tilde{t}_s,\tilde{t}_e)\|_{rt}$$
$$\{\|[t_s,t_e)\|_{rt}\} = \{[\|t_s\|_{rt},\|t_e\|_{rt})\}$$
$$\{\|[t_s,t_e)\|_{rt}\} = \{\|[t_s,t_e)\|_{rt}\}$$

The equivalence for the intersection and the union function can be proven analogously. □

Intuitively, the result of the difference $[t_s,t_e) - [\tilde{t}_s,\tilde{t}_e)$ on ongoing intervals is the following set of (ongoing interval, RT)-pairs: the result is equivalent to input time interval $[t_s,t_e)$ at the reference times when the two input time intervals do not overlap and it is equivalent to the sub-time intervals $[t_s,\tilde{t}_s)$ and $[\tilde{t}_e,t_e)$ at the reference times when the input time intervals overlap. As a numeric example, the difference $[01/14,10/20) - [08/17,now)$ is equivalent to the following (ongoing interval, RT)-pairs according to Table 3.1 in Theorem 2:

VT	**RT**
$[01/14,10/20)$	$\neg([01/14,10/20)$ overlaps $[08/17,now))$
	$=\{(-\infty,08/18)\}$
$[01/14,07/18)$	$[01/14,10/20)$ overlaps $[08/17,now))$
	$=\{[08/18,\infty)\}$
$[now,10/20)$	$[01/14,10/20)$ overlaps $[08/17,now))$
	$=\{[08/18,\infty)\}$

3.6 Functions in Relational Algebra Operators

This section discusses the storage of function results, i.e., (ongoing interval, RT)-pairs, in ongoing relations. Ongoing relations associate each tuple with a single reference time attribute *RT*. We discuss a general mechanism that guarantees the correct restriction of a tuple's single *RT* attribute when multiple or nested interval functions are used in a relational algebra operator.

The result of functions on ongoing intervals are (ongoing interval, RT)-pairs: the ongoing interval is associated with the reference time *RT* when the time interval is part of the result. We store the function result in an ongoing relation as follows: Each (ongoing interval, RT)-pair is stored in its own result tuple. The ongoing time interval is stored in an attribute corresponding to the

function; the reference time of the tuple is restricted with the pair's reference time. This ensures that the tuple belongs to the instantiated relations only at the reference times when the ongoing interval is part of the function result. As an example, the result of projection with difference function, $\pi_{\mathbf{B},VT-T}(\mathbf{R})$, is equivalent to the following ongoing relation:

$$\begin{aligned}&\pi_{\mathbf{B},VT-T}(\mathbf{R})\\&\quad\equiv\{(r.\mathbf{B},T_r,RT)\mid\exists r\in\mathbf{R}((T_r,RT_r)\in(r.VT-T)\wedge RT=(r.RT\wedge RT_r))\}\end{aligned}$$

The ongoing relation includes the ongoing interval T_r as a separate attribute. The reference time of the tuple is set to $r.RT\wedge RT_r$, i.e., the intersection of the original reference time of tuple r and the reference time RT_r when the time interval T_r is part of the difference result.

Multiple and nested functions are flattened into nested projections so that each projection list includes one interval function at most. As base cases for the flattening ϕ we define $\phi(\pi_A(\mathbf{R}))=\pi_A(\mathbf{R})$ if A is a list of attribute names, and $\phi(\pi_{*,A}(\mathbf{R}))=r$ if A is an attribute name. The auxiliary function $a(t)$ constructs a unique identifier out of the attribute names and constants (mapped to numerical identifiers) in term t, e.g., $a(c-(d-5))=cd1$. We write $\pi_{X/C}$ to rename X to C. With this, the flattening of a projection with multiple or nested difference functions is defined as:

$$\begin{aligned}&\phi(\pi_{t_1,\ldots,t_i-t_j,\ldots,t_n}(\mathbf{R}))\\&\quad=\phi(\pi_{t_1,\ldots,a(t_i-t_j),\ldots,t_n}(\pi_{*,a(t_i)-a(t_j)/a(t_i-t_j)}(q)))\\&\quad\text{where } p=\phi(\pi_{*,t_i}(\mathbf{R}))\text{ and }q=\phi(\pi_{*,t_j}(p))\end{aligned}$$

Expressions p and q (line 3) flatten the input arguments of the interval function and the interval function is evaluated in a projection with flattened input arguments (line 2). The other terms in the input argument are recursively flattened in the enclosing projection.

Examples for flattening multiple and nested functions in a projection are:

- $\phi(\pi_{c-d,e}(\mathbf{R}))=\pi_{cd,e}(\pi_{*,c-d/cd}(\mathbf{R}))$
- $\phi(\pi_{g-h,c-d,e}(\mathbf{R}))=\pi_{gh,cd,e}(\pi_{*,c-d/cd}(\pi_{*,g-h/gh}(\mathbf{R})))$
- $\phi(\pi_{g,c-(d-e)}(\mathbf{R}))=\pi_{g,cde}(\pi_{*,c-de/cde}(\pi_{*,d-e/de}(\mathbf{R})))$

For interval functions with several result pairs, the flattening is equivalent to calculating the cross product of the result intervals in case of multiple functions and it is equivalent to applying the enclosing function to each result interval in case of nested functions.

A single reference time attribute is sufficient to incorporate the reference time restrictions of all interval functions. We explain this for the case of multiple interval functions. One interval function on an input tuple results in several tuples that, together, represent the function result at every reference time of the input tuple's reference time RT. For multiple interval functions, we compute the cross product of the result tuples of each function for an input tuple and restrict the reference time of a result tuple to the common reference times. This ensures that we do not lose the result of a function at any reference time. Instead, at each reference time of the input tuple's RT value, the combined result tuples whose RT attribute contains the reference time represent the results of all the interval functions.

To evaluate multiple and nested interval functions in other relational algebra operators, the flattening described in the context of the projection is used. A selection with functions is evaluated by first flattening these functions in the context of a projection and then referencing the functions' attributes in the selection (t_i is a term and $a(t_i)$ the auxiliary function to construct an attribute identifier):

$$\begin{aligned}&\sigma_{\theta(t_1,\ldots,t_m,r)}(\mathbf{R})\\&\quad\equiv \pi_{\mathbf{R}.*}(\sigma_{\theta(a(t_1),\ldots,a(t_m),r)}(\phi(\pi_{*,t_1,\ldots,t_m}(\mathbf{R}))))\end{aligned}$$

Example 13. *The following selection applied to relation* $\mathbf{E}$ *of our running example is evaluated as*

$$\begin{aligned}&\sigma_{[07/16,09/18)\ during\ (VT-[08/17,+10/20))}(\mathbf{E})\\&\quad\equiv \pi_{Name,Role,VT}(\sigma_{[07/16,09/18)\ during\ Diff}(\\&\qquad\qquad \pi_{Name,Role,VT,VT-[08/17,+10/20)/Diff}(\mathbf{E})))\end{aligned}$$

The innermost projection flattened the difference as defined. The enclosing selection references attribute Diff *that contains the difference result interval in its predicate. The selection selects the tuples with the predicate as known. The outermost projection ensures that the schema of the result relation is equal to the schema of the input relation; the attribute* Diff *with the difference result is omitted.*

3.7 Implementation

For ongoing data types (time points and time intervals), the *RT* attribute, and predicates on ongoing data types, we use the implementation provided by [MB20b]. An ongoing time point $a+b$ is implemented as a structure composed of two fixed time points. A tuple's reference time *RT* is implemented as a list of maximal, non-overlapping fixed time intervals. We use this representation also for the result of a predicate, i.e., the reference times when the predicate is *true*.

We implement the functions given in Theorem 2 as set-returning functions (cf. Algorithm 3). A set-returning function is a function that can return multiple, composite values and returns one of these values per invocation. Multiple return values are needed since both the difference and the union function return three values for each input. Composite return values are beneficial since a function determines the result time interval *and* the reference times when the time interval is part of the result.

Algorithm 3 shows our implementation of the difference $[t_s,t_e) - [\tilde{t}_s,\tilde{t}_e)$ as set-returning function *diff*. The result is a record of the form *(diffVT, diffRT)*: *diffVT* refers to the result time interval and *diffRT* to the reference times when the time interval is part of the result.

Procedure: Difference function diff$([t_s,t_e),[\tilde{t}_s,\tilde{t}_e))$
Input: Function context *fctx*
Output: A single record of the form *(diffVT, diffRT)* or ω

1 Copy variables of *fctx* to local;
2 **if** *first call* **then**
3 $\quad b_{ovlp} \leftarrow [t_s,t_e)$ overlaps $[\tilde{t}_s,\tilde{t}_e)$;
4 $\quad b_{notOvlp} \leftarrow \neg b_{ovlp}$;
`// List of the result records.`
5 $\quad res \leftarrow [([t_s,t_e), b_{notOvlp}),$
$\qquad ([t_s,\tilde{t}_e), b_{ovlp}),$
$\qquad ([\tilde{t}_e,t_e), b_{ovlp})]$;
6 **end**
7 **if** $size(res) > 0$ **then**
8 $\quad out \leftarrow res$.pop(0);
9 **else**
10 $\quad out \leftarrow \omega$;
11 **end**
12 Copy local variables to *fctx*;
13 **return** *out*;

Algorithm 3: Difference function on ongoing intervals.

At each invocation of *diff*, either a single record of the form *(diffVT, diffRT)* is returned or ω is returned to indicate the end of the function. The multiple invocations of the function are natively handled by PostgreSQL. The input is a function context, *fctx*, that keeps variables between different invocations. The function context stores the list of result records, *res*, that need to be returned at future invocations. At the first invocation of *diff*, the reference times when the input time intervals overlap, b_{ovlp}, and the reference times when they do not overlap, $b_{notOvlp}$, are calculated. We perform the calculation only once and reuse the reference times for the result records. The list of result records, *res*, is initialized with the three possible result records (cf. Theorem 2). At each invocation, one of the result records is removed from *res* and returned.

We illustrate the evaluation of $diff([10/17, now), [01/14, 10/20))$ with its four invocations: three to return the result records and one to notify the system that all result records have been returned. The variables b_{ovlp} and $b_{notOvlp}$ are initialized to $b_{ovlp} = [[10/18, \infty)]$ and $b_{notOvlp} = [(-\infty, 10/18)]$. The list of result records is set to $res = [([10/17, now), [(-\infty, 10/18)]), ([10/17, 01/14), [[10/18, \infty)]), ([10/20, now), [[10/18, \infty)])]$. The first record in *res* is removed and $out = ([10/17, now), [(-\infty, 10/18)])$ is returned. In the second invocation, the next record in *res* is removed and $out = ([10/17, 01/14), [[10/18, \infty)])$ is returned. In the third invocation, the last record in *res* is removed and $out = ([10/20, now), [[10/18, \infty)])$ is returned. In the forth invocation, *res* is empty and ω is returned. This signals the system that all result records have been returned and there will be no more invocations of the function.

We illustrate the usage of the set-returning function in SQL with the projection $\pi_{B_1, B_2, VT_x - VT_y}(\mathbf{R})$. The corresponding *select* statement is as follows:

```
SELECT *
FROM (SELECT B1, B2, diffVT, (diffRT ∧ RT) as RT
      FROM (SELECT B1, B2,(diff(VTx, VTy)).*, RT
            FROM R) as R1) as R2
WHERE cardinality(RT) > 0;
```

The inner-most select query uses the set-returning function $diff(VT_x, VT_y)$ (cf. Algorithm 3) to calculate the difference. The returned result records of the form (*diffVT*, *diffRT*) are decomposed with the $.*$ syntax into its two attributes *diffVT* and *diffRT*. In the enclosing select query, the result time interval is stored in the *diffVT* attribute and the reference time of the result tuple is set to the conjunction of the reference times in *diffRT* and the input tuple's reference time *RT*. Finally, the result tuples with an empty reference time are discarded (cardinality$(RT) > 0$).

Optimization We discuss the optimization of the intersection and the difference function. The optimization of the union function resembles the one of the difference function.

The intersection function for ongoing intervals returns a single ongoing time interval, which is the result at all reference times. We optimize its implementation by using a standard function (not set-returning) that returns the ongoing time interval in Theorem 2. This is possible since restricting a result tuple's reference time with the trivial reference time $\{(-\infty,\infty)\}$ does not change the result tuple's reference time. The benefit of this optimization is that we do not need to restrict the result tuple's reference time, resulting in an improved runtime of the intersection function.

Our optimization for the difference function focuses on minimizing the number of result records for any given input. The size of the optimized result gets close to the result size for the difference function on fixed time intervals (cf. Section 3.8.2). This increases the performance of relational algebra operators that are applied afterwards to the result relation. The optimization also illustrates that handling ongoing values easily gets complex (cf. Algorithm 5 and Algorithm 6). Clearly, it is important that interval functions transparently handle ongoing values on behalf of applications.

The implementation of the difference function in Algorithm 3 returns three result records independent of the input intervals. Depending on the input intervals, the difference result can be represented with less result records. Algorithm 4 shows our optimized implementation of the difference function, $\mathit{diffOpt}([t_s,t_e),[\tilde{t}_s,\tilde{t}_e))$. The optimized implementation differs from the simple implementation in Algorithm 3 in how the list of result records, *res*, is determined. The key idea is to merge the time intervals of the three possible result records (cf. Algorithm 3) into two or even one time interval. First, the algorithm attempts to merge the records with overlapping reference time, $([t_s,\tilde{t}_s),b_{ovlp})$ and $([\tilde{t}_e,t_e),b_{ovlp})$ into one record. This is shown in Algorithm 5: the two time intervals can be merged into one time interval if they are non-empty at disjoint reference times, i.e., at each reference time in b_{ovlp}, the result can be represented with a single time interval, and the merged interval is a valid ongoing time interval, i.e., the start and end point are ongoing time points $a{+}b \in \Omega$ with $a \leq b$. Afterwards, the algorithm attempts to merge the resulting record(s) *candOvlp* of the previous step with the result record $([t_s,t_e),\neg b_{ovlp})$ with non-overlapping reference times. This is shown in Algorithm 6: if $([t_s,t_e),\neg b_{ovlp})$ and a result record of *candOvlp* instantiate to the same time interval at the reference times when $\neg b_{ovlp}$ changes from *true* to *false* (or vice versa), the two records can be merged.

Procedure: Difference function diffOpt$([t_s,t_e),[\tilde{t}_s,\tilde{t}_e))$
Input: Function context *fctx*
Output: A single record of the form *(diffVT, diffRT)* or ω

```
1  Copy variables of fctx to local;
2  if first call then
3      b_ovlp ← [t_s,t_e) overlaps [t̃_s,t̃_e);
4      b_notOvlp ← ¬b_ovlp;
       // Get minimal list of result records.
5      if b_ovlp = ∅ then
           // b_notOvlp = [(−∞,∞)].
6          res ← [([t_s,t_e), b_notOvlp)]
7      else if b_notOvlp = ∅ then
           // b_ovlp = [(−∞,∞)].
           // Try merging [t_s,t̃_s) and [t̃_e,t_e).
8          res ← mergeOvlpRecords([t_s,t̃_s),[t̃_e,t_e));
9      else
           // Try merging [t_s,t̃_s) and [t̃_e,t_e).
10         resCandidates ← mergeOvlpRecords([t_s,t̃_s),[t̃_e,t_e));
           // Try merging with [t_s,t_e).
11         res ← mergeWithNotOvlpRecord([t_s,t_e),[t̃_s,t̃_e),resCandidates);
12     end
13 end
14 if size(res) > 0 then
15     out ← res.pop(0);
16 else
17     out ← ω;
18 end
19 Copy local variables to fctx;
20 return out;
```

Algorithm 4: Difference function on ongoing intervals with minimal number of result records.

The predicates used in the three algorithms are evaluated at most once among all algorithms. For instance, the same *overlaps* predicate is used in all three algorithms, but is only evaluated at the first occurrence and its result boolean is passed by reference to the other algorithms. This is done to reduce the runtime overhead added by the evaluation of predicates on ongoing values (cf. the experimental evaluation by Mülle et al. [MB20b]).

Procedure: mergeOvlpRecords($[t_s,\tilde{t}_s),[\tilde{t}_e,t_e)$)
Input: $[t_s,\tilde{t}_s)=[a+b,\tilde{a}+\tilde{b})$ and $[\tilde{t}_e,t_e))=[\tilde{c}+\tilde{d},c+d)$: the two ongoing intervals to be merged
Output: A list of (ongoing interval, RT)-records

1 $b_{ovlp} \leftarrow [t_s,t_e)$ overlaps $[\tilde{t}_s,\tilde{t}_e)$;
2 **if** $b_{ovlp} \wedge t_s < \tilde{t}_s = \emptyset$ **then**
  // $[t_s,\tilde{t}_s)$ empty at all *rt* in b_{ovlp}.
3 **return** $[([\tilde{t}_e,t_e),b_{ovlp})]$;
4 **else if** $b_{ovlp} \wedge \tilde{t}_e < t_e = \emptyset$ **then**
  // $[\tilde{t}_e,t_e)$ empty at all *rt* in b_{ovlp}.
5 **return** $[([t_s,\tilde{t}_s),b_{ovlp})]$;
6 **else if** $b_{ovlp} \wedge t_s < \tilde{t}_s \wedge \tilde{t}_e < t_e = \emptyset$ **then**
  // $[t_s,\tilde{t}_s)$ and $[\tilde{t}_e,t_e)$ might be merged.
7 $[[x,y)] \leftarrow b_{ovlp} \wedge t_s < \tilde{t}_s$;
8 **if** $x = -\infty$ **then**
9 *mergedInterval* $\leftarrow [a+\tilde{d},\tilde{a}+d)$;
10 **else**
  // $y = \infty$
11 *mergedInterval* $\leftarrow [\tilde{c}+b,c+\tilde{b})$;
12 **end**
13 **if** *mergedInterval is an ongoing interval* **then**
14 **return** $[(mergedInterval,b_{ovlp})]$;
// Merging not possible.
15 **return** $[([t_s,\tilde{t}_s),b_{ovlp}),([\tilde{t}_e,t_e),b_{ovlp})]$;

Algorithm 5: Function that merges the two result records $[t_s,\tilde{t}_s)$ and $[\tilde{t}_e,t_e)$ into one ongoing time interval.

3.8 Evaluation

This section compares runtime and result size of our solution with a solution that is only applicable to fixed time intervals and the state-of-the-art solution for ongoing intervals from Torp et al. [TJS04]. The evaluation considers the three standard functions for time intervals and we vary the percentage of ongoing intervals, the duration of fixed time intervals, the number of input tuples, and the number of functions in a query.

3.8.1 Setup

The empirical evaluation is conducted on a 3.40 GHz machine with 16GB main memory and an SSD. The client and the database server run on the same machine. We use the PostgreSQL 9.4.0

Procedure: mergeWithNotOvlpRecord($[t_s,t_e), [\tilde{t}_s,\tilde{t}_e), candOvlp$)
Input: $[t_s,t_e) = [a+b, c+d)$, $[\tilde{t}_s,\tilde{t}_e) = [\tilde{a}+\tilde{b}, \tilde{c}+\tilde{d})$: two ongoing intervals,
candOvlp: list of the (time interval, RT)-records for the overlapping case
Output: A list of (ongoing interval, RT)-records

1 $b_{ovlp} \leftarrow [t_s,t_e)$ overlaps $[\tilde{t}_s,\tilde{t}_e)$;
2 $[[x,y)] \leftarrow \neg b_{ovlp}$;
3 **if** $size(candOvlp) = 1$ **then**
4 $\quad ([m+n, o+p), \_) \leftarrow candOvlp[0]$;
5 $\quad$ **if** $x = -\infty \wedge \|[t_s,t_e)\|_{y-1} = \|[m+n,o+p)\|_{y-1}$ **then**
6 $\quad\quad$ **return** $[([a+n, c+p), [(-\infty,\infty)])]$;
7 $\quad$ **else if** $y = \infty \wedge \|[t_s,t_e)\|_x = \|[m+n,o+p)\|_x$ **then**
8 $\quad\quad$ **return** $[([m+b, o+d), [(-\infty,\infty)])]$;
9 $\quad$ **else if** $\|[t_s,t_e)\|_x = \|[m+n,o+p)\|_x \wedge \|[t_s,t_e)\|_{y-1} = \|[m+n,o+p)\|_{y-1}$ **then**
10 $\quad\quad$ **return** $[([m+n, o+p), [(-\infty,\infty)])]$;
11 **else**
$\quad$ // $candOvlp = [([t_s,\tilde{t}_s), b_{ovlp}), ([\tilde{t}_e,t_e), b_{ovlp})]$.
12 $\quad$ **if** $x = -\infty \wedge \|[t_s,t_e)\|_{y-1} = \|[t_s,\tilde{t}_s)\|_{y-1}$ **then**
13 $\quad\quad$ **return** $[([t_s, c+\tilde{b}), [(-\infty,\infty)]), ([\tilde{t}_e,t_e), b_{ovlp}))]$;
14 $\quad$ **else if** $y = \infty \wedge \|[t_s,t_e)\|_x = \|[\tilde{t}_e,t_e)\|_x$ **then**
15 $\quad\quad$ **return** $[([\tilde{c}+b, t_e), [(-\infty,\infty)]), ([t_s,\tilde{t}_s), b_{ovlp}))]$;
16 $\quad$ **else if** $\|[t_s,t_e)\|_x = \|[\tilde{t}_e,t_e)\|_x \wedge \|[t_s,t_e)\|_{y-1} = \|[t_s,\tilde{t}_s)\|_{y-1}$ **then**
17 $\quad\quad$ **return** $[([\tilde{c}+b, c+\tilde{b}), [(-\infty,\infty)]), ([a+\tilde{d}, \tilde{a}+d), b_{ovlp}))]$;
18 **end**
// Not mergeable.
19 **return** $candOvlp \cup [([t_s,t_e), \neg b_{ovlp})]$;

Algorithm 6: Function that merges the result records of Algorithm 5 for $rt \in b_{ovlp}$ and the result record $[t_s,t_e)$ for $rt \in \neg b_{ovlp}$ into results records for all rt.

kernel extended with our implementation of the intersection, difference, and union function for ongoing intervals.

Table 3.2 summarizes the real-world and synthetic data sets. As real-world data set, we use *MozillaBugs* [LPD13] that records the history of bugs in the Mozilla project. We use the following two relations: (1) **BugInfo** records general information about a bug: ID, product, component, operating system, severity, and valid time. Bugs that have not been resolved as of the date of the data export have ongoing valid time intervals. (2) **BugAssignment** records the email address of the person assigned to a bug, the bug ID, and the valid time.

Figure 3.5 shows the distribution of the start points of the ongoing intervals in *MozillaBugs*. 50% of the tuples with ongoing intervals in relations **BugInfo** and **BugAssignment** are located within the last two years of the history. For experiments with an increasing number of tuples

Table 3.2: Characteristics of the experiment data sets.

(a) Real-world data set.

	MozillaBugs	
	BugInfo B	**BugAssignment A**
Cardinality	394,878	582,668
# ongoing	60,372 (15%)	63,588 (11%)
Time intervals	$\{[a,now),[c,d)\}$	$\{[a,now),[c,d)\}$
Duration of $[c,d)$	∅ 215 days	∅ 199 days
Time span	20 years	20 years

(b) Synthetic data sets.

	$\mathbf{D^{dur}}$	$\mathbf{D^{per}}$	$\mathbf{D^{nest}}$
Cardinality	9M	9M	1M
# ongoing	$0\% \times 100\%$	0% - 100%	20%
Time intervals	$\{[c,d)\} \times \{[a,now)\}$	$\{[a,now),[c,d)\}^2$	$\{[a,now),[c,d)\}^7$
Duration of $[c,d)$	25 - 150 days	10 days	1 - 50 days
Time span	225 - 350 days	210 days	250 days

we grow the size of the real-world data set by growing the history of **BugInfo** backward and use all records in **BugAssignment** that match to the bug IDs in **BugInfo**. This means that the percentage of ongoing intervals decreases as the data size grows.

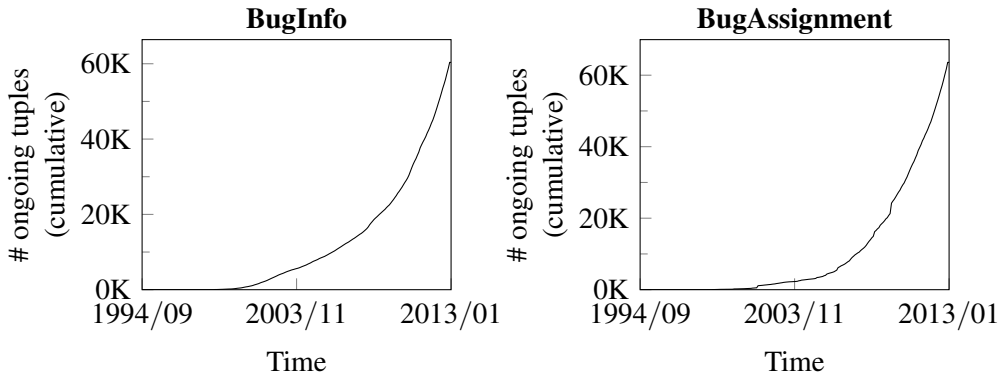


Figure 3.5: Start point distribution of ongoing intervals.

We compare our approach to an approach that is only applicable to fixed time intervals but guarantees expected result intervals and to Torp's approach [TJS04], which is applicable to ongoing intervals, but does not guarantee a result with the expected intervals at every reference time. We refer to the former as *fixed* and to the latter as *torp*. We implemented both approaches as set-returning functions in the PostgreSQL 9.4.0 kernel. Note that the implementation natively provided by PostgreSQL for the three standard functions allows returning only one time interval.

For Torp's approach, we implemented the two provided functions, intersection and difference, according to the case distinctions in [TJS04].

On the synthetic data sets, we use a projection that applies a function to the two valid time attributes of the input relation: $Q_f^{\pi} = \pi_{\mathrm{f}(T_1,T_2)}(\mathbf{R})$. Input relation $\mathbf{R}$ is the result of a pre-computed join. On the real-world data set *MozillaBugs*, we use two join queries. Query $Q_{\mathrm{diff}}^{\bowtie}(\mathbf{B},\mathbf{A})$ determines for a bug with severity 'major' when a person assigned to the bug was not working on it:

$$Q_{\mathrm{diff}}^{\bowtie}(\mathbf{B},\mathbf{A}) = \pi_{\mathbf{B}.*,\mathbf{B}.VT\ -\ \mathbf{A}.VT}(\mathbf{B} \bowtie_{\mathbf{B}.ID=\mathbf{A}.ID \wedge \text{Severity='major'}} \mathbf{A})$$

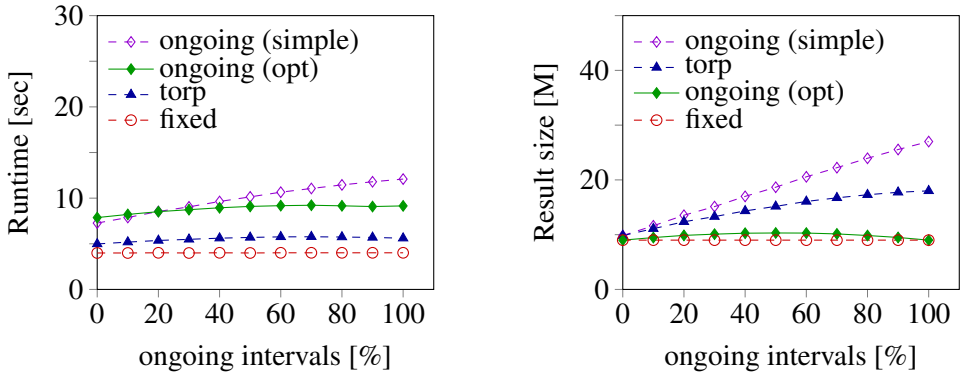
$Q_f^{\bowtie}(\mathbf{B})$ determines the result of function f on the self-join of relation $\mathbf{B}$ on similar bugs. Similar bugs are bugs that affect the same product, component, and operating system (θ_{sim}):

$$Q_f^{\bowtie}(\mathbf{B}) = \pi_{\mathbf{B}.*,\mathbf{B}'.*,f(\mathbf{B}.VT,\mathbf{B}'.VT)}(\mathbf{B} \bowtie_{\theta_{sim} \wedge \mathbf{B}.\text{Severity='major'}} \mathbf{B}')$$

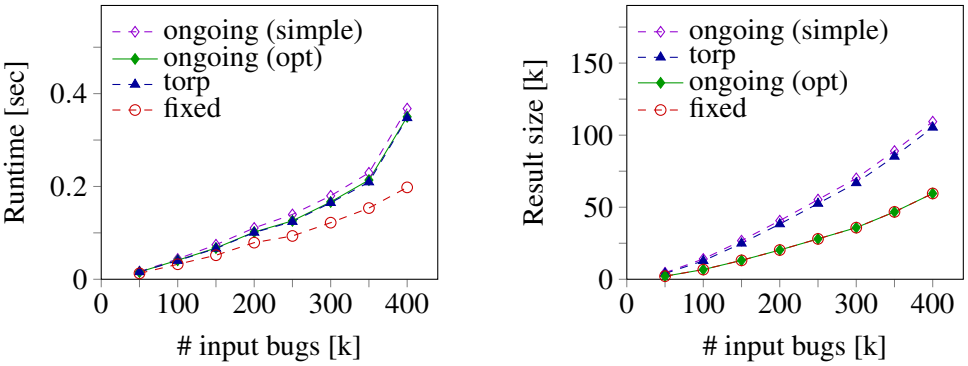
3.8.2 Optimization

We evaluate the impact of the optimization of the difference function in Algorithm 4 on runtime and result size. We use the synthetic data set D^{per}, which is a self-join of a table with a varying percentage of ongoing intervals. We calculate the difference function between each pair of time intervals. Figure 3.6a shows that the simple implementation's runtime and result size both increase linearly with an increase in ongoing percentage. For the optimized implementation, these remain almost constant, with a slight symmetrical increase towards a peak at 50%, which happens because the combination of fixed and ongoing intervals (rather than both being ongoing, or both being fixed) has the most potential to lead to the worst-case of requiring three result tuples even in the optimized implementation. Compared to the fixed implementation, we do see that there is an overhead in the runtime, but that the result size overhead is neglible. Torp's approach has a faster runtime than the ongoing approach, but carries a result size overhead that behaves more closely to that of the unoptimized difference function than that of the optimized variant. These findings can also be replicated on real-world data. Figure 3.6b shows that as the number of input bugs grows, the result size of the optimized approach remains close to that of the fixed approach, whereas both the simple implementation and Torp's approach grow at a faster rate.

Due to these advantages, all following sections only consider the optimized implementation for the ongoing approach.



(a) Projection Q^{π}_{diff} on D^{per}.



(b) Join $Q^{\bowtie}_{\text{diff}}(\mathbf{B},\mathbf{A})$ on *MozillaBugs*.

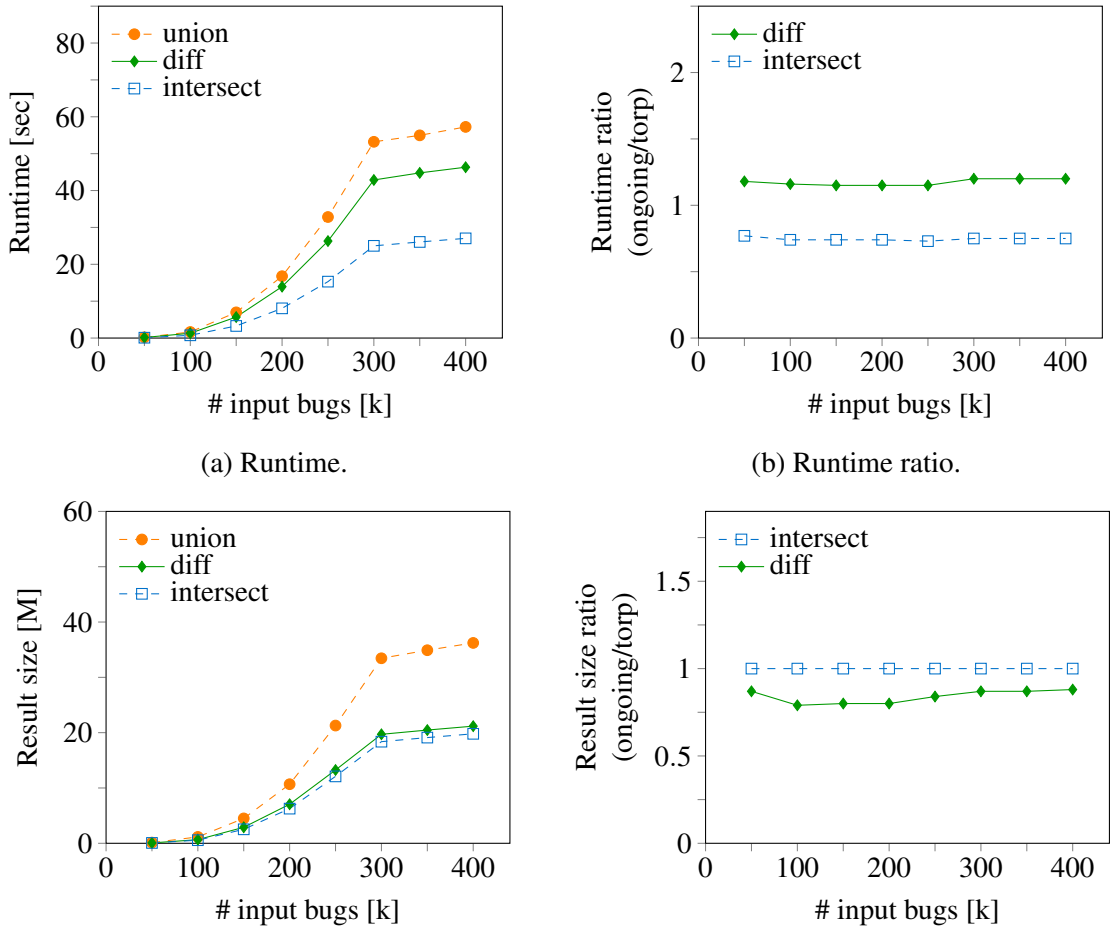
Figure 3.6: Optimization of the difference function.

3.8.3 Functions

We evaluate how the runtime and result size of the three standard functions for time intervals compare. Figure 3.7a for runtime and Figure 3.7c for result size show that the intersection function is the fastest since it always returns only one tuple and is not implemented as a set-returning function. The difference function returns more tuples and has the additional runtime overhead of set-returning functions. The union function is slower and returns larger results because it does not have the same optimizations in its implementation that the difference function has. Compared to Torp, we see a similar runtime behavior in Figure 3.7b, with the difference function having a slight overhead since it also needs to compute the reference times when the ongoing interval is

part of the result whereas Torp does not have to. In Figure 3.7d, the result size is exactly the same for the intersection (since each approach always returns one time interval per input tuple), and slightly less for our approach than for Torp since the ongoing approach can represent the result time intervals with less ongoing intervals than Torp in some cases, but never more.

Figure 3.7 shows a similar pattern in each function, where runtime and result size rise linearly until around 300k input bugs, after which the slope flattens significantly. When increasing the number of input bugs, the percentage of calculating the difference between fixed and ongoing intervals decreases, resulting in fewer result tuples and a faster computation.



(a) Runtime.

(b) Runtime ratio.

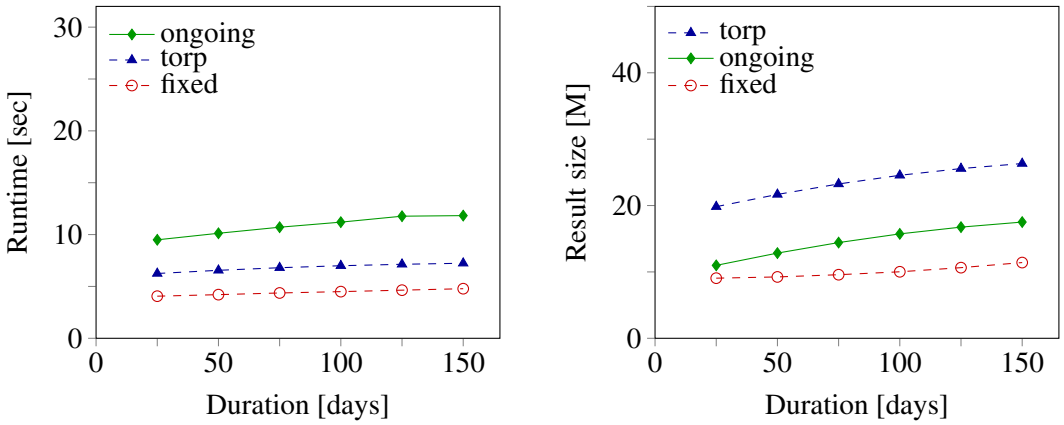
(c) Result size.

(d) Result size ratio.

Figure 3.7: Different functions on *MozillaBugs* ($Q_f^{\bowtie}(\mathbf{B})$).

3.8.4 Properties of Time Intervals

The main properties of time intervals that have an impact on the result size and runtime are the percentage of ongoing intervals and the duration of the fixed time intervals. We discussed the influence of the percentage in Section 3.8.2. Longer durations of fixed intervals make it more likely for each pair of intervals to overlap and more occurences of the worst case of the difference function, in which three result tuples are returned. The data set D^{dur} for this experiment is a cross-join between a table containing a constant number of fixed tuples and a table containing a constant number of ongoing tuples. The start dates are randomized (in a 200-day range), and for the fixed tuples the duration is selected randomly up to a maximum duration. Figure 3.8a shows that the runtime of the ongoing approach increases, though sublinearly, as the maximum duration increases. For the result size, we observe that the ongoing approach leads to more tuples than the fixed approach, but less than in Torp's approach (Figure 3.8b).



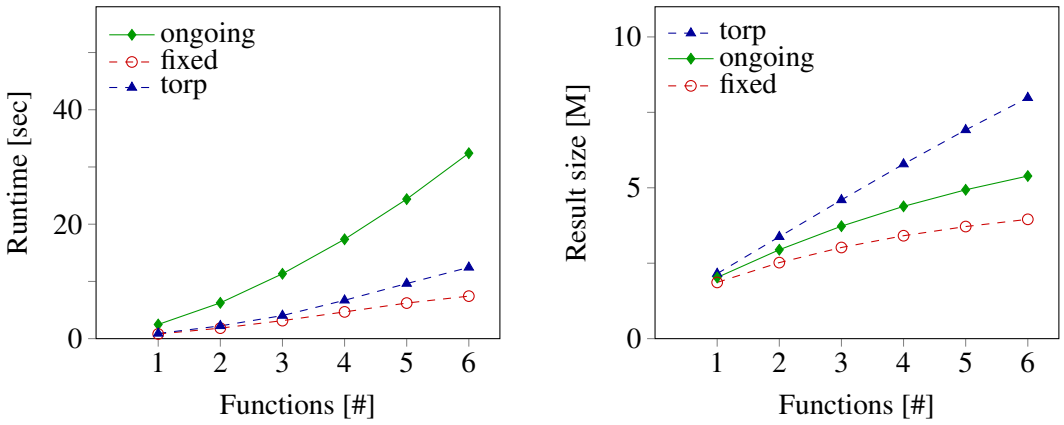
(a) Runtime. (b) Result size.

Figure 3.8: Varying duration (Q^{π}_{diff} on D^{dur}).

3.8.5 Scalability

We evaluate how the nesting depth of functions affect the result size and runtime. For this experiment, we generated a data set D^{nest} with a fixed number of tuples, with each tuple having six additional valid time attributes; this is equivalent to adjusting the valid time of a tuple with the valid times of six tuples in a temporal operator (cf. Section 3.2). These additional valid time intervals, of which 20% are ongoing, have randomized starting points and durations, but are guaranteed to overlap with the first valid time VT. In the first nesting depth, we calculate

the difference between VT and the first additional valid time, in the second nesting depth we calculate the difference between the output of the first nesting and the second additional VT, and so on. In Figure 3.9b, we observe that the result size rises slightly sublinearly with the number of nested functions. Since the runtime cost of each nesting depth is primarily determined by the number of result tuples from the previous nesting depth, this result size essentially acts as a derivative of the runtime, resulting in the runtime that we see in Figure 3.9a. This shows that our approach that evaluates functions to results that remain valid as time passes by scales beyond simple queries to support complex nested queries as well.



(a) Runtime. (b) Result size.

Figure 3.9: Scalability with number of functions on D^{nest}.

3.9 Conclusions

We propose the first approach that evaluates the standard functions for ongoing intervals, namely, intersection, difference, and union, to results that remain valid as time passes by. At each reference time, the function result consists of the expected time intervals. This is necessary to get the correct truth value for predicates on the result time intervals and for temporal relational algebra operators that require the expected intervals for their correctness. We propose function results that are pairs consisting of an ongoing interval and the reference times when this ongoing interval is part of the result. To store the results of interval functions in databases, we leverage ongoing relations with a single reference time attribute that integrates the restrictions from the results of all interval functions.

As future work, we plan to add support for additional functions on ongoing data types like the duration function whose result are ongoing integers. We want to propose an aggregation operator for ongoing relations and determine an efficient grouping in the presence of ongoing attributes (equality at some reference times only) and the reference time attribute (tuples belong to relations at different reference times).

CHAPTER 4

Aggregation on Ongoing Relations

Abstract

Relational aggregation is an important operator to summarize data. It divides the tuples in a relation into groups and aggregates each group into a single tuple. Data with ongoing time points are organized in ongoing relations. In an ongoing relation, the ongoing values of a tuple and the set of tuples in the relation change by time passing by. This adds new challenges to the aggregation task. A tuple might belong to different aggregation groups at disjoint times. Thus, the aggregate of a group at one time does not remain valid for other times.

This paper proposes the first solution that evaluates the aggregation operator on ongoing relations to results that remain valid as time passes by. The idea is to first determine the fixed groups of tuples with equal fixed grouping attributes (attributes without ongoing values), then divide each fixed group into ongoing groups according to the ongoing grouping attributes (attributes with ongoing values) and finally aggregate each ongoing group into a single result tuple. Our approach incrementally calculates the groups and their aggregate values without materializing the tuples that belong to the group. We describe the seamless integration of the aggregation operator on

ongoing relations in PostgreSQL and analytically and empirically evaluate its runtime, result size, and memory requirements.

4.1 Introduction

The aggregation operator is a widely used operator in databases. It allows organizing data and retrieving interesting, statistical information about the data. Data with ongoing values add new challenges to the aggregation task since ongoing values change as time passes by and so does the data aggregation.

Data with ongoing values are organized in ongoing relations. In an ongoing relation, tuples consist of fixed and ongoing attributes and the reference time attribute. The values of fixed attributes remain the same as time passes by whereas the values of ongoing attributes change depending on the time. The reference time attribute contains the reference times when *now* can be instantiated in the tuple and the tuple belongs to the relation. The goal is to evaluate the aggregation operator on ongoing relations to results that remain valid as time passes by. Formally, given an ongoing relation $\mathbf{R}$ and aggregation ${}_{\mathbf{G}}\vartheta_{\phi}$ with grouping attributes $\mathbf{G}$ and the set of aggregate functions ϕ, we want to determine an aggregation result ${}_{\mathbf{G}}\vartheta_{\phi}(\mathbf{R})$, such that at every possible reference time rt, the aggregation result is equivalent to the result obtained by instantiating *now* in $\mathbf{R}$ and evaluating the aggregation on the instantiated relation: $\forall rt(\|{}_{\mathbf{G}}\vartheta_{\phi}(\mathbf{R})\|_{rt} \equiv {}_{\mathbf{G}}\vartheta_{\phi}(\|R\|_{rt}))$. The bind operator $\|\cdot\|_{rt}$ replaces all occurrences of *now* with the reference time rt.

The aggregation operator determines the aggregate values based on groups of tuples with equal grouping attribute values. The key challenge is to determine these groups: (1) since ongoing values change as time passes by, tuples can be equal at some reference times only, and (2) tuples with equal grouping attribute values might belong to the relation at different reference times. The first case occurs when the grouping attributes contain ongoing attributes; the second case occurs for ongoing relations due to the presence of the *RT* attribute. As a consequence, a tuple might be part of a group at some reference times only and it might be part of different groups at disjoint reference times.

Example 14. *Consider a consulting company that runs software development projects. An employee has a fixed-term or a permanent employment. Fixed-term employments have fixed start points that indicate the start of the employment and fixed end dates that indicate the end of the employment. Permanent employments have fixed start dates but end dates that keep increasing*

until the contract is modified. These end dates are ongoing. *Projects are conducted for a fixed timeframe: a project has a fixed start date and a fixed end date. Employees get assigned to various projects while they are employed. A selected relation of our running example is shown in Figure 4.1 and discussed below.*

P

	PID	Name	VT	RT
p_1	500	Ann	$[03/08, +08/23)$	$\{[03/09, \infty)\}$
p_2	500	Bob	$[03/08, 08/23)$	$\{(-\infty, \infty)\}$
p_3	501	Eve	$[05/14, +10/29)$	$\{[05/15, \infty)\}$
p_4	501	John	$[05/14, +10/29)$	$\{[05/15, \infty)\}$

Figure 4.1: Ongoing relation.

Relation **P** *lists selected project assignments. An assignment is described by the project id* PID*, the* Name *of the employee that is assigned to the project, the valid time VT when the employee is assigned to the project, and the reference time RT. For instance, tuple p_1 records that employee Ann is assigned to a project with id 500. The assignment is valid from* $03/08$ *until no later than* $08/23$ *and the tuple belongs to the relation from reference time* $03/09$ *on.*

To improve the effectiveness and quality of how software is developed in the company, the company wants to try out various software processes and study their impact on effectiveness and quality. For the studies, the company wants to find suitable projects. A suitable project should have many employees with equal assignment timeframes in the project, so that the set of employees that participate in the study for one software process remains stable during the study. The following aggregation query retrieves this information:

$$\mathbf{V} \leftarrow {}_{PID,VT}\vartheta_{count(*)}(\mathbf{P})$$

The aggregation uses fixed attribute PID *and ongoing attribute* VT *as grouping attributes. Figure 4.2 illustrates the input tuples of relation* **P** *and the aggregation result, $\mathbf{V} = \{a_1, a_2, a_3, a_4\}$. A tuple's reference time RT is shown along the x-axis and the tuple's valid time is shown along the y-axis at the tuple's reference time RT. For instance, result tuple a_3 records that 2 employees are assigned to project 500 from* $03/08$ *until* $08/23$ *exclusively and that the tuple belongs to the relation from reference time* $08/23$ *on.*

The aggregation groups depend on the reference time: the values of ongoing grouping attributes are equal at some reference times only and tuples belong to the relation at different reference times. An example of the former are the aggregation groups of tuples p_1 and p_2. The grouping attribute values of tuples p_1 and p_2 are not equal *up to reference time* $08/22$*, resulting in*

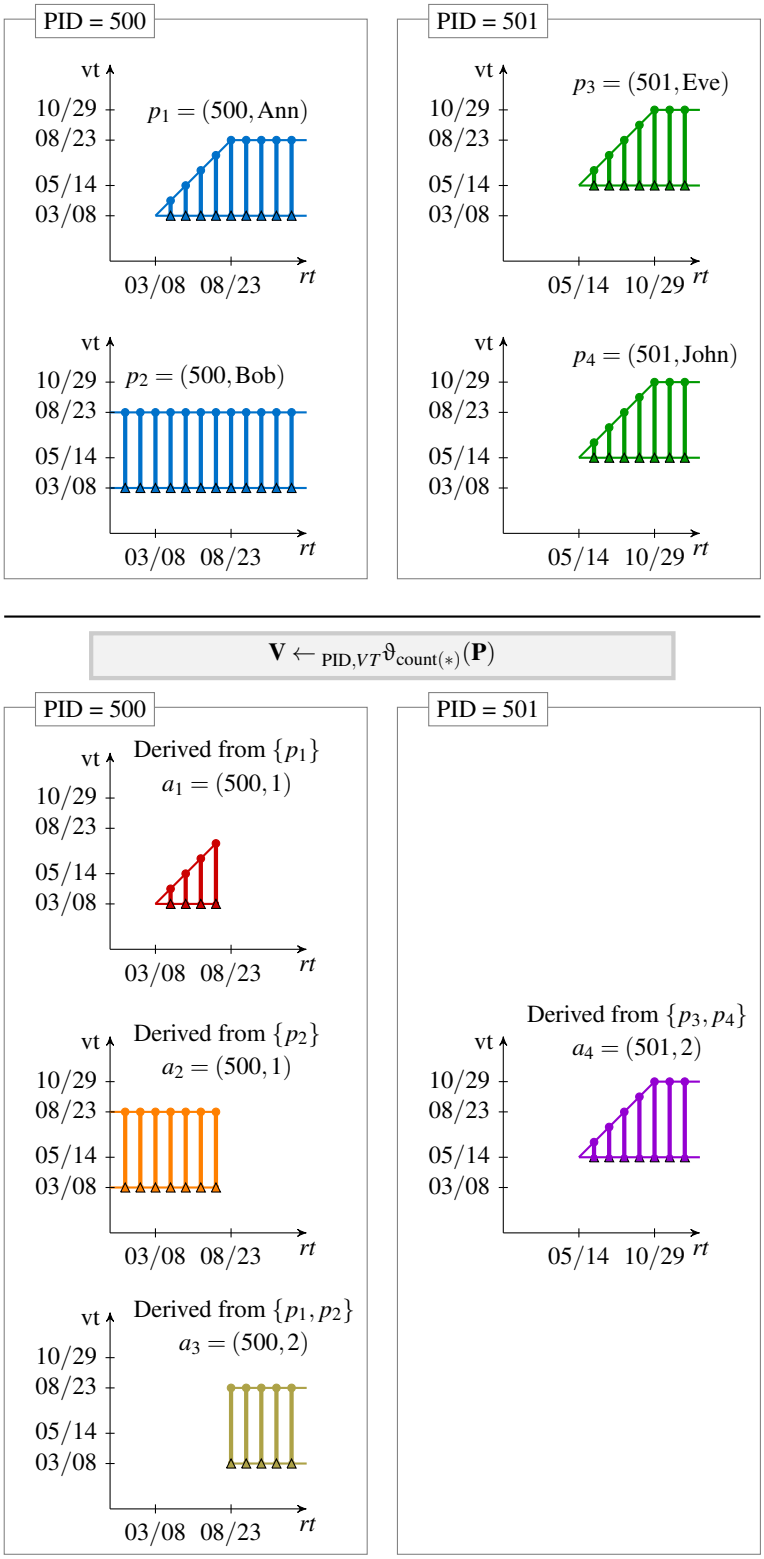


Figure 4.2: The aggregation result remains valid.

separate aggregation groups for p_1 and p_2 and separate result tuples a_1 and a_2. The grouping attribute values of tuples p_1 and p_2 are equal *from reference time* $08/23$ *on, resulting in a common aggregation group and result tuple a_3. An example of tuples belonging to the relation at some reference times only is tuple p_3. Tuple p_3 does not belong to the relation up to reference time* $05/14$ *and thus, is not part of any aggregation group up to reference time* $05/14$. *From reference time* $05/15$ *on, tuple p_3 belongs to the relation and is part of the aggregation group that produces result tuple a_4.*

Observe that the aggregation result remains valid as time passes by. At each reference time, the aggregation groups and thus, the aggregation result are equivalent to the groups and aggregation result, respectively, determined by the conventional aggregation operator. The resulting ongoing relation consists of the expected aggregation results at every reference time. For instance, at reference time $08/23$*, both Ann and Bob are assigned to the project with id 500 with common assignment timeframe* $[03/08, 08/23)$ *and both Eve and John are assigned to the project with id 501 with common assignment timeframe* $[05/14, 08/23)$. *The expected aggregation result at reference time* $08/23$ *consists of two tuples: the first tuple recording that two employees are assigned to the project with id 500 during* $[05/14, 08/23)$ *and the second tuple recording that two employees are assigned to the project with id 501 during* $[05/14, 08/23)$. *This is consistent with the ongoing result relation: at reference time* $08/23$*, tuples a_3 and a_4 describe the aggregation result since reference time* $08/23$ *is only contained in $a_3.RT$ and $a_4.RT$. The two tuples record exactly the expected aggregation result for reference time* $08/23$.

We propose a solution that evaluates the aggregation operator on ongoing relations to results that remain valid as time passes by. The focus is to determine the correct groups for a tuple depending on the reference time. Our solution splits the reference time value of the input tuples, such that we get groups of tuples with equal grouping attribute values and the same reference time RT. All attributes of an ongoing relation (fixed and/or ongoing) can be used as grouping attributes.

We consider aggregate functions on fixed attributes only; aggregate functions on ongoing attributes are beyond the scope of this paper and part of future research. All aggregate functions that are supported by current database systems can also be used with our solution. Examples are *count*, *min*, *max*, *sum*, and *avg*.

Our solution provides an aggregation algorithm for ongoing relations that is integrated in the query processing pipeline of the PostgreSQL database system. The key idea of our aggregation algorithm is to first group tuples into fixed groups according to the fixed grouping attributes,

then to group the tuples of a fixed group into ongoing groups according to the ongoing grouping attributes and the reference time attribute RT. The ongoing groups are then the input to calculate the aggregate values. The aggregation results is an ongoing relation that includes a single result tuple for each ongoing group. The three-step algorithm allows us to leverage the existing, optimized strategies of database systems for grouping fixed values and for the incremental calculation of aggregate values.

The aggregation algorithm incrementally builds the fixed and ongoing groups and incrementally calculates the aggregate values. This allows the algorithm to never materialize the tuples that belong to a group. This keeps the memory consumption low, especially since a tuple might belong to several ongoing groups. The aggregation algorithm first sorts the input tuples according to the fixed grouping attribute. This allows the algorithm to process tuples that belong to the same fixed group consecutively without the need of storing the tuples of a fixed group. Then, the algorithm incrementally builds the ongoing groups for a fixed group. An ongoing group is represented with (1) a single master tuple that provides the fixed and ongoing attribute values and the reference time RT of the group and with (2) the aggregate values that are incrementally calculated. Ongoing groups are incrementally built by manipulating the group's reference time RT. The algorithm starts with an ongoing group that is represented by the first tuple in the fixed group. Each subsequent tuple t_i in the fixed group then splits an ongoing group into two ongoing groups: the first ongoing group g_1 that consists of the reference times when the group and the tuple are equal according to the ongoing grouping attributes and have common reference times RT and the second ongoing group g_2 whose reference time is the reference time of the original ongoing group minus the reference time of g_1. A new ongoing group g_3 with tuple t_i as its master tuple is created for the reference times when the tuple belongs to the relation and is not equal to any ongoing group. These reference times are included in the group's RT value. Whenever a tuple is conceptually added to an ongoing group (groups g_1 and g_3), the aggregate values of the group are updated. Once all tuples of a fixed group have been processed, the aggregate values of the ongoing groups are finalized and a result tuple for each ongoing group is produced. After these result tuples are produced, the algorithm starts processing the tuples of the next fixed group.

Our contributions are the following:

- We propose an aggregation algorithm for ongoing relations that yields results that remain valid as time passes by. The algorithm splits the RT value of the input tuples, such that we get groups of tuples with equal grouping attribute values and the same reference time

RT. The aggregation result is an ongoing relation with a single, aggregated tuple for each group.

- The runtime of the algorithm scales, in its worst case, quadratically with its output, i.e. the number of ongoing groups. In practical usage, we observe only a linear scaling.

- The number of ongoing groups in a fixed group scales quadratically with the size of the fixed group. An aggregation resulting in larger fixed group sizes conversely implies a smaller number of fixed groups by the same factor. Thus, the effective overall complexity with respect to the fixed group size is linear instead of quadratic.

- The algorithm avoids materializing the tuples that belong to an aggregation group by incrementally calculating the groups and their aggregate values. The memory requirements per ongoing group remain constant independent of the group's size.

- Our aggregation algorithm incorporates existing, optimized strategies of the database system to determine groups and aggregate values. We use these grouping mechanisms to determine tuples with equal fixed grouping attribute values. The aggregate calculation strategies are used to determine the aggregate values of each ongoing group.

- We describe the seamless integration of the aggregation operator on ongoing relations in PostgreSQL. The integration supports grouping according to fixed and ongoing attributes.

The paper is organized as follows. Section 4.2 discusses related work. Section 4.3 introduces preliminaries. Section 4.4 formalizes the aggregation on ongoing relations and Section 4.5 discusses and analyzes our implementation of the aggregation in the PostgreSQL database system. Section 4.6 describes the empirical evaluation of our solution. Section 4.7 concludes the paper and points to future research.

4.2 Related Work

The most commonly used ongoing time point is *now*. The SQL-92 standard [MS93] includes the reserved keywords CURRENT_TIME, CURRENT_DATE, and CURRENT_TIMESTAMP that denote the ongoing time point *now* for different time granularities. Existing database

systems cannot store ongoing time points. They instantiate ongoing time points immediately at compile time when statements are issued. Various research approaches propose solutions for adding support for ongoing time points and time intervals to database systems [CDI+97, DJTS09, APS+16, APT16, TJS00, FM96, JL01, TJB97, STS03, SST09, Sno87].

In the following, we will discuss research approaches that proposed solutions for evaluating queries on relations with ongoing time points [TJS04, CDI+97, ASTS13, MB20b].

Torp et al. [TJS04] propose an approach for modifications on temporal databases with ongoing time points. Torp et al. show that performing temporal modifications on tuples that are instantiated when accessed leads to incorrect modifications and thus, incorrect data in the database. For evaluating queries on relations with ongoing time points, Torp et al. use the framwork proposed by Clifford et al. [CDI+97, DJTS09], which instantiates *now* whenever it is accessed during query processing. Thus, queries are evaluated on instantiated relations without ongoing time points. The advantage is that the existing aggregation for relations without ongoing time points can be used. The disadvantage is that the aggregation result is only valid at the chosen reference time and the result gets invalidated by time passing by.

Anselma et al. [ASTS13] propose a basic temporal algebra for relations with ongoing time points and have extended their approach to support indeterminacy for tuples with *now* [APS+16]. Their approach uses functions on ongoing time intervals to adjust the valid time of the input tuples for the result. Ongoing time points are kept uninstantiated in the query result whenever their time domain can represent the adjusted valid time. If not, their approach replaces *now* with the reference time. This yields results that get invalidated by time passing by. Anselma et al. consider only relations without a reference time attribute RT and they have not worked out how predicates on ongoing values, including the *equals* predicate, are defined and evaluated. Their approach cannot support aggregation with ongoing grouping attributes on ongoing relations, which include a reference time attribute.

Mülle et al. [MB20b] propose a framework for evaluating functions, predicates, and relational algebra operators on ongoing data types to results that remain valid as time passes by. The key idea is to evaluate the operations at every reference time and represent the results as ongoing data types. Mülle et al. propose a relational algebra with the five basic operators: selection, projection, Cartesian product, union, and difference. As the result of relational algebra operators, Mülle et al. introduce ongoing relations in which each tuple is associated with a reference time attribute RT. The value of the RT attribute consists of the reference times when the tuple can be instantiated. Predicates in relational algebra operators select tuples by restricting a tuple's reference time

to the reference times when the predicate is *true*. Mülle et al. have extended their approach to support functions on ongoing time intervals in relational algebra operators [MB20a]. Their approaches do not consider extended relational algebra operators like aggregation on ongoing relations. Specifically for the aggregation on ongoing relations, Mülle et al. did not work out how to group tuples in the presence of ongoing attributes (equality at some reference times only) and the reference time attribute (a tuple belongs to the instantiated relation at some reference times only).

Temporal relational algebra operators [DBGJ16, BJ09] adjust the valid time [JS09] of the input tuples for the result tuples. Various approaches for relations without ongoing time points and without a reference time attribute *RT* have been proposed [BJ02, ABPT01, DBGJ16, Tom98, Tom96, BJS00, LM97, DDL02]. Their common idea is to split the valid times of a group of tuples into sub-time intervals, such that they are either identical or disjoint [ABPT01, DBGJ16, Tom98]. The idea of the valid-time splitting is similar to how our approach determines the aggregation groups with equal fixed and ongoing attribute values. Our approach considers ongoing relations, i.e., relations with ongoing time points and a reference time attribute *RT*. Our approach splits the reference time *RT* of the tuples in a group with equal fixed grouping attribute values into sub-reference times, such that the sub-*RT*s are either identical or disjoint (cf. Section 4.4). For the splitting, our approach considers the *RT* of the tuples in conjunction with the reference times when the tuples are equal.

4.3 Preliminaries

We assume a linearly ordered, discrete time domain $\mathscr{T}$ with $-\infty$ as the lower limit and ∞ as the upper limit. Fixed data types consist of values that do not change as time passes by. Examples include strings, integers, and time points of $\mathscr{T}$. A fixed time interval $[t_s, t_e) \in \mathscr{T} \times \mathscr{T}$ consists of an inclusive fixed start point and an exclusive fixed end point. Ongoing data types consist of values that change as time passes by. Ongoing values can be instantiated to a fixed value with the bind operator $\|\cdot\|_{rt}$ at a reference time rt. Composite ongoing values are instantiated by instantiating each component. The time domain for ongoing time points is $\Omega = \{a{+}b | a, b \in \mathscr{T} \wedge a \leq b\}$ [MB20b]. Ongoing time point $a{+}b$ instantiates to the following fixed time point

at reference time $rt \in \mathscr{T}$:

$$\|a{+}b\|_{rt} = \begin{cases} a & rt \leq a \\ rt & a < rt < b \\ b & \text{otherwise} \end{cases}$$

Ongoing time point $a{+}b$ subsumes fixed time points $a \equiv a{+}a$, time point $now \equiv -\infty{+}\infty$, growing time points $a{+} \equiv a{+}\infty$, and limited time points ${+}b \equiv -\infty{+}b$. An ongoing time interval $[t_s, t_e)$ is a closed-open time interval with ongoing start and end points. At each reference time rt, an ongoing interval instantiates to a fixed time interval by instantiating its start and end point: $\|[t_s, t_e)\|_{rt} = [\|t_s\|_{rt}, \|t_e\|_{rt})$.

$R = (\mathbf{A})$ denotes the schema of a fixed relation $\mathbf{R}$ with fixed attributes $\mathbf{A} = A_1, \ldots, A_n$. A tuple r with schema R is a finite list that contains for every A_i a value from the domain of A_i. A relation $\mathbf{R}$ over schema R is a finite set of tuples over R. The schema of an ongoing relation [MB20b] consists of fixed and ongoing attributes $A_1, \ldots, A_n$ and the reference time attribute RT: $R = (A_1, \ldots, A_n, RT)$. An ongoing relation $\mathbf{R}$ is instantiated to a fixed relation with the bind operator $\|\mathbf{R}\|_{rt}$ at a reference time rt: $\|\mathbf{R}\|_{rt} = \{(\|r.A_1\|_{rt}, \ldots, \|r.A_n\|_{rt}) | r \in \mathbf{R} \wedge rt \in r.RT\}$. $r.A_i$ denotes the value of attribute A_i in tuple r. The value of the reference time attribute RT is a set of reference time points and denotes when the attribute values of a tuple can be instantiated. At all other reference times, the tuple is discarded. In a base ongoing relation $\mathbf{S}$, each tuple $s \in \mathbf{S}$ can be instantiated at all reference times, i.e., $s.RT = \{(-\infty, \infty)\}$. The representation of the RT attribute is not relevant for the semantics since it is an internal attribute that cannot be used in functions and predicates. $\theta(r)$ denotes the application of predicate θ to tuple r.

Operations on ongoing data types evaluate to results that remain valid as time passes by: at each reference time rt, the result of an operation op on ongoing values is equivalent to the result obtained by first instantiating the ongoing input values $i_1, \ldots, i_n$ at reference time rt and then evaluating the corresponding operation op^F for fixed data types:

$$\forall rt \in \mathscr{T}(\|op(i_1, \ldots, i_n)\|_{rt} \equiv op^F(\|i_1\|_{rt}, \ldots, \|i_n\|_{rt}))$$

We use the ^{F}-superscript to refer to an operation on fixed data types. Operations on fixed data types retain their standard behavior. The aggregation operator for fixed relations is defined as follows with $\mathbf{G}$ as the fixed grouping attributes and $\phi = [\phi_1, \ldots, \phi_k]$ as the list of aggregate functions

over $\mathbf{R}.\mathbf{A}$:

$$_{\mathbf{G}}\vartheta^{F}_{\phi}(\mathbf{R}) = \{(\mathbf{g}, v_1, \ldots, v_k) | \mathbf{g} \in \pi^{F}_{\mathbf{G}}(\mathbf{R}) \wedge v_1 = \phi_1(\sigma^{F}_{\mathbf{G}=\mathbf{g}}(\mathbf{R})) \\ \wedge \cdots \wedge v_k = \phi_k(\sigma^{F}_{\mathbf{G}=\mathbf{g}}(\mathbf{R}))\}$$

Predicates on ongoing values change their truth value, i.e., *true* or *false*, depending on the reference time. We represent the result of a predicate as the reference times when the predicate is *true*. For instance, $([03/08, +08/23) = [03/08, 08/23)) \equiv \{[08/23, \infty)\}$, i.e., the predicate is *true* from reference time $08/23$ on. The conjunction of two predicates is *true* at the reference times when both predicates are *true*; the negation of a predicate is *true* at the reference times when the predicate is *false*, i.e., at all reference times in $\mathscr{T}$ except the reference times when the predicate is *true*.

4.4 Aggregation

This section formalizes the aggregation operator on ongoing relations, such that its results remain valid as time passes by.

The tuples of an ongoing relation [MB20b] consist of fixed and ongoing attributes and the reference time attribute RT. Fixed attributes have fixed data types whereas ongoing attributes have ongoing data types. The reference time of a tuple includes the reference times when the values of the tuple can be instantiated and thus, the tuple belongs to the instantiated relation.

Example 15. *Consider ongoing relation* $\mathbf{P}$ *in Example 14. A tuple consists of the fixed attributes* PID *and* Name *and the ongoing attribute* VT. *As an example, the reference time of tuple* p_1*,* $p_1.RT = \{[03/09, \infty)\}$*, states that the values of the tuple's fixed and ongoing attributes can be instantiated from reference time* $03/09$ *on and that tuple* p_1 *belongs to the instantiated relations from reference time* $03/09$ *on.*

Definition 7. (Aggregation $_{\mathbf{G}}\vartheta_{\phi}(\mathbf{R})$ on ongoing relations) *Let* $\mathbf{R}$ *be an ongoing relation with schema* $(\mathbf{A}, RT)$*, let* $\mathbf{G} \subseteq \mathbf{A}$ *be the grouping attributes and let* ϕ *be a set of aggregate functions over fixed attributes of* $\mathbf{R}$*. Let* ϑ^{F} *be the aggregation operator for fixed relations. The aggregation* $_{\mathbf{G}}\vartheta_{\phi}(\mathbf{R})$ *on ongoing relation* $\mathbf{R}$ *is defined as*

$$_{\mathbf{G}}\vartheta_{\phi}(\mathbf{R}) = \mathbf{V} \textit{ iff } \forall rt \in \mathscr{T}(\|\mathbf{V}\|_{rt} = {}_{\mathbf{G}}\vartheta^{F}_{\phi}(\|\mathbf{R}\|_{rt}))$$

The result of the aggregation on ongoing relations is an ongoing relation **V**, such that, at each reference time *rt*, result ongoing relation **V** is equal to the relation obtained by evaluating the aggregation for fixed relations on the instantiated input relation. This yields results that remain valid as time passes by.

In the following, we first introduce the two auxiliary functions *canon* and *splitRT* that we use to define the result relation of the aggregation operator on ongoing relations. Then, we define this result relation in Theorem 3, such that it satisfies Definition 7 of the aggregation operator.

Given a set of reference times RT, two ongoing values that differ syntactically might be semantically equal at the reference times in RT. For instance, consider ongoing time intervals $[03/08, +08/23)$ and $[03/08, 08/23)$ at the reference times in $RT = \{[08/23, \infty)\}$. Both ongoing time intervals instantiate to time interval $[03/08, 08/23)$ at the reference times in RT and are semantically equal although their representation differs. In order to be able to apply syntactical equality to the ongoing values, auxiliary function $canon(v, RT)$ canonicalizes the representation of an ongoing value v depending on a set of reference times RT.

Definition 8. (Canonicalization canon(v, RT)) *Let v be an ongoing value and $\mathscr{D}$ be its domain. Let RT be a set of reference times. Then, $canon(v, RT)$ determines a representation of v in $\mathscr{D}$ that is the same for all values in $\mathscr{D}$ that are equal to v at the reference times in RT:*

$$\begin{aligned} &\forall v \in \mathscr{D}(\forall rt \in RT(\|canon(v, RT)\|_{rt} = \|v\|_{rt})) \\ &\wedge \forall v_1, v_2 \in \mathscr{D}\ (canon(v_1, RT) = canon(v_2, RT) \\ &\qquad \Leftrightarrow (\forall rt \in RT(\|v_1\|_{rt} =^F \|v_2\|_{rt}))) \end{aligned}$$

We use this function in Theorem 3 to determine a canonicalized representation of the values of the ongoing grouping attributes for the tuples that belong to the same ongoing group. Together with *splitRT* in Definition 9, this allows us to determine the ongoing groups in a fixed group with a simple projection and retrieve the tuples of an ongoing group with only a selection to aggregate these tuples into a single result tuple. This is similar to how the aggregation operator for fixed relations determines groups and aggregates them (cf. Section 4.3). For our implementation in Section 4.5.2, we use the attribute values of the tuple that creates an ongoing group as the *canonicalized attribute values* for this group. The tuples that belong to this group are then incrementally determined by semantically comparing their attribute values with the canonicalized attribute values of the group.

Example 16. *Consider the three ongoing time intervals* $T_1 = [03/08, 08/23)$, $T_2 = [03/08, +08/23)$, $T_3 = [03/08, now)$. *We assume one possible instance of function canon that satisfies Definition 8.*

At all reference times in $RT_1 = \{[08/23, \infty)\}$, T_1 *and* T_2 *are* equal *and* T_3 *is* not equal *to time intervals* T_1 *and* T_2. *Given* RT_1, *function* canon *returns the same time interval for* T_1 *and* T_2 *and a different time interval for* T_3*:*

$$canon([03/08, 08/23), \{[08/23, \infty)\}) = [03/08, 08/23)$$
$$canon([03/08, +08/23), \{[08/23, \infty)\}) = [03/08, 08/23)$$
$$canon([03/08, now)), \{[08/23, \infty)\}) = [03/08, now)$$

Consider reference time $RT_2 = \{[03/08, 08/23)\}$. *At all reference times in* RT_2, T_2 *and* T_3 *are equal and* T_1 *is not equal to time intervals* T_2 *and* T_3. *Given* RT_2, *function* canon *returns the same time interval for* T_2 *and* T_3 *and a different time interval for* T_1*:*

$$canon([03/08, 08/23), \{[03/08, 08/23\}) = [03/08, 08/23)$$
$$canon([03/08, +08/23), \{[03/08, 08/23\}) = [03/08, now)$$
$$canon([03/08, now)), \{[03/08, 08/23\}) = [03/08, now)$$

Definition 9. (RT-splitter splitRT($RT_x, \mathbf{Z}$)) *Let* RT_x *be an RT value, i.e., a set of reference times, and let* $\mathbf{Z}$ *be a finite set of RT values. Then,* $splitRT(RT_x, \mathbf{Z})$ *splits reference time* RT_x *into sub-RT values depending on the RT values in* $\mathbf{Z}$*:*

$$RT \in splitRT(RT_x, \mathbf{Z}) \Leftrightarrow$$
$$(1)\ RT \subseteq RT_x \wedge \forall RT_z \in \mathbf{Z}(RT \cap RT_z = \emptyset \vee RT \subseteq RT_z)$$
$$\wedge\ (2)\ \nexists RT' \supset RT\,((1)\ is\ fulfilled\ for\ RT')$$

A split *RT* value (1) is contained in the reference time value RT_x and is either contained or disjoint from the *RT* values in $\mathbf{Z}$ and (2) is maximal, i.e., there does not exist a superset of the split *RT* value that fulfills the condition in (1).

We use function *splitRT* in Theorem 3 to ensure that the tuples that belong to the same ongoing group have the same reference time RT. Together with the canonicalization of the ongoing attribute values (Definition 8), this allows us to retrieve for a fixed group the ongoing groups and their tuples with simple projection and selection in order to correctly aggregate each group, similar to how the aggregation on fixed relations divides a relation into groups and aggregates each group (cf. Section 4.3). For our implementation in Section 4.5.2, we split the reference time RT of a tuple on-the-fly with predicates on the ongoing grouping attributes. Note that a predicate on ongoing values evaluates to the set of reference times that includes the reference times when the predicate is *true*. The on-the-fly splitting is possible since we incrementally create and update ongoing groups based on the tuples that have been processed so far. When all input tuples have been processed, the RT values of the ongoing groups to which an input tuple belongs satisfy the constraints imposed by Definition 9 for the reference time RT_x of the tuple.

Example 17. *We assume a possible instance of splitRT function that satisfies Definition 9.*

Consider $RT_x = \{[03/09, \infty)\}$ *and* $\mathbf{Z} = \{\{[08/23, \infty)\}\}$. *The result of* $splitRT(RT_x, \mathbf{Z})$ *consists of the following split RT values:*

$$splitRT(RT_x, \mathbf{Z}) = \{\{[03/09, 08/23)\}, \{[08/23, \infty)\}\}$$

The two RT values are contained in RT_x *and maximal. The split RT value* $\{[03/09, 08/23)\}$ *is disjoint from the RT value in* $\mathbf{Z}$*; the split RT value* $\{[08/23, \infty)\}$ *is contained in the RT value in* $\mathbf{Z}$.

The aggregation operator groups tuples with equal grouping attribute values together and aggregates each group into a single result tuple. In an ongoing relation, the group of equal tuples differs depending on the reference time: (1) tuples with equal grouping attribute values might belong to the relation at different reference times and (2) tuples are equal at some reference times only since the ongoing grouping attribute values change as time passes by.

Theorem 3. *Let* $\mathbf{R}$ *be an ongoing relation with schema* $(\mathbf{B}, \mathbf{G}_f, \mathbf{G}_o, RT)$. *Let* $\mathbf{G}_f$ *be the fixed grouping attributes and* $\mathbf{G}_o$ *be the ongoing grouping attributes. Let* canon *and* splitRT *be as given in Definition 8 and Definition 9, respectively. Let* $\phi = [\phi_1, \ldots, \phi_k]$ *be a list of aggregate functions over fixed attributes in* $\mathbf{R}$. *The result relation of aggregation* ${}_{\mathbf{G}_f, \mathbf{G}_o}\vartheta_{\phi}(\mathbf{R})$ *in Definition 7*

is equivalent to the following ongoing relation:

$$
\begin{aligned}
&{}_{\mathbf{G}_f,\mathbf{G}_o}\vartheta_{\phi}(\mathbf{R}) \\
&\quad\equiv \{(g_f, g_o, v_1, \ldots, v_k, RT_o) | g_f \in \pi^F_{\mathbf{G}_f}(\mathbf{R}) \wedge \mathbf{X} = \sigma_{\mathbf{G}_f = g_f}(\mathbf{R}) \\
&\qquad\wedge \mathbf{Z} = \{RT | \exists x_1, x_2 \in \mathbf{X}(RT = (x_1.\mathbf{G}_o = x_2.\mathbf{G}_o \\
&\qquad\qquad\qquad\qquad \wedge x_1.RT \wedge x_2.RT))\} \\
&\qquad\wedge \mathbf{O} = \{(x.\mathbf{B}, g_f, g_o, RT) | x \in \mathbf{X} \\
&\qquad\qquad\qquad \wedge g_o = canon(x.\mathbf{G}_o, RT) \\
&\qquad\qquad\qquad \wedge RT \in splitRT(x.RT, \mathbf{Z})\} \\
&\qquad\wedge (g_o, RT_o) \in \pi_{\mathbf{G}_o}(\mathbf{O}) \wedge v_1 = \phi_1(\sigma^F_{\mathbf{G}_o = \mathbf{g}_o \wedge RT = RT_o}(\mathbf{O})) \\
&\qquad\wedge \cdots \wedge v_k = \phi_k(\sigma^F_{\mathbf{G}_o = \mathbf{g}_o \wedge RT = RT_o}(\mathbf{O}))\}
\end{aligned}
$$

The result ongoing relation consists of an aggregate tuple for each ongoing group. The second line determines the fixed groups, i.e., the groups of tuples with the same fixed grouping attribute values. For a fixed group g_f, its ongoing groups are determined and stored in $\mathbf{O}$. All tuples in an ongoing group have the same, canonicalized values of the ongoing grouping attributes (line 6) and the same reference time RT (line 7). This is ensured by splitting the reference time of a tuple with the reference time values in $\mathbf{Z}$, which result from the reference times when any two tuples in $\mathbf{X}$ have equal ongoing grouping attribute values and common RT (lines 3 - 4). The last two lines determine the result of the aggregate functions for an ongoing group.

Proof. We prove the equivalence in Theorem 3 by showing that the definition of the aggregation for ongoing relations in Definition 7 holds for $\mathbf{V}$ as the ongoing relation in Theorem 3:

$$\forall rt \in \mathscr{T}(\|\mathbf{V}\|_{rt} = {}_{\mathbf{G_f},\mathbf{G}_o}\vartheta^F_{\phi}(\|\mathbf{R}\|_{rt}))$$

For a reference time $rt \in \mathscr{T}$, we show that ongoing result relation $\mathbf{V}$ contains (1) the same fixed grouping attribute values, (2) for each fixed grouping attribute value the same ongoing grouping attribute values, and (3) for each grouping attribute value the same aggregate values as the *expected result relation* obtained by evaluating the aggregation operator for fixed relations on the instantiated input relation.
Ongoing projection $\pi_{\mathbf{C}}(\mathbf{S})$ on ongoing relation $\mathbf{S}$ returns tuples with schema $(\mathbf{C}, RT)$; fixed projection $\pi^F_{\mathbf{C}}(\mathbf{S})$ on ongoing relation $\mathbf{S}$ returns tuples with schema $(\mathbf{C})$.

Let $g_f \in \pi^F_{\mathbf{G}_f}(\mathbf{R})$. Ongoing relations $\mathbf{X} = \sigma_{\mathbf{G_f}=g_f}(\mathbf{R})$ and $\mathbf{O}$ consist of the same tuples at reference time rt: $\|\mathbf{X}\|_{rt} = \|\mathbf{O}\|_{rt}$. The tuples in $\mathbf{X}$ and $\mathbf{O}$ differ in the representation of the values of the ongoing grouping attributes $\mathbf{G}_o$ and the reference time RT. Function *splitRT* in Definition 8 ensures that exactly for the reference times in $x.RT$, there exist $RT \in \text{splitRT}(x.RT, \_)$ that contain these reference times (split RT values are contained in $x.RT$ and maximal): $rt \in x.RT \Leftrightarrow \exists RT \in \text{splitRT}(x.RT, \_)(rt \in RT)$. Function *canon* in Definition 8 ensures that $\|\text{canon}(x.\mathbf{G}_o, \_)\|_{rt} = \|x.\mathbf{G}_o\|_{rt}$. Thus, $\|\mathbf{X}\|_{rt} = \|\mathbf{O}\|_{rt}$.

Step 1: Same values of the fixed grouping attribute $\mathbf{G}_f$.

$$\begin{aligned} &\|\pi_{\mathbf{G}_f}(\mathbf{V})\|_{rt} \stackrel{?}{=} \pi^F_{\mathbf{G}_f}({}_{\mathbf{G_f},\mathbf{G}_o}\vartheta^F_\phi(\|\mathbf{R}\|_{rt})) \\ \Leftrightarrow\ &\|\pi_{\mathbf{G}_f}(\mathbf{V})\|_{rt} \stackrel{?}{=} \pi^F_{\mathbf{G}_f}(\|\mathbf{R}\|_{rt}) \end{aligned}$$

According to the definition of the aggregation operator for fixed relations (cf. Section 4.3), the fixed grouping attribute values of ${}_{\mathbf{G_f},\mathbf{G}_o}\vartheta^F_\phi(\|\mathbf{R}\|_{rt})$ result from input relation $\|\mathbf{R}\|_{rt}$. We show in the following that the equality holds. The fixed grouping attribute values in $\mathbf{V}$ result from $\pi^F_{\mathbf{G}_f}(\mathbf{R})$. For $g_f \in \pi^F_{\mathbf{G}_f}(\mathbf{R})$,

$$\begin{aligned} \|\pi_{\mathbf{G}_f}(\sigma_{\mathbf{G}_f=g_f}(\mathbf{V}))\|_{rt} &= \|\pi_{\mathbf{G}_f}(\mathbf{O})\|_{rt} \\ &= \|\pi_{\mathbf{G}_f}(\mathbf{X})\|_{rt} \\ &= \|\pi_{\mathbf{G}_f}(\sigma_{\mathbf{G}_f=g_f}(\mathbf{R}))\|_{rt} \\ &= \pi^F_{\mathbf{G}_f}(\sigma^F_{\mathbf{G}_f=g_f}(\|\mathbf{R}\|_{rt})) \end{aligned}$$

The $\mathbf{G}_f$ values in $\sigma_{\mathbf{G}_f=g_f}(\mathbf{V})$ result from $\mathbf{O}$. The ongoing selection σ and the ongoing projection π remain valid as time passes by [MB20b] and $\|g_f\|_{rt} = g_f$.

The $\mathbf{G}_f$ values in $\pi^F_{\mathbf{G}_f}(\mathbf{R})$ are a superset of the $\mathbf{G}_f$ values in $\pi^F_{\mathbf{G}_f}(\|\mathbf{R}\|_{rt})$, $\pi^F_{\mathbf{G}_f}(\mathbf{R}) \supseteq \pi^F_{\mathbf{G}_f}(\|\mathbf{R}\|_{rt})$: the fixed projection omits the RT attribute and $\|\pi^F_{\mathbf{G}_f}(\mathbf{R})\|_{rt} = \pi^F_{\mathbf{G}_f}(\mathbf{R})$; the bind operator on ongoing relation $\mathbf{R}$ can only discard but never add tuples. Then, two cases for the $\mathbf{G}_f$ values can occur.

Case 1: $g_f \in \pi^F_{\mathbf{G}_f}(\mathbf{R})$ and $g_f \in \pi^F_{\mathbf{G}_f}(\|\mathbf{R}\|_{rt})$
$\sigma^F_{\mathbf{G}_f=g_f}(\|\mathbf{R}\|_{rt})$ is non-empty and thus, $\|\sigma_{\mathbf{G}_f=g_f}(\mathbf{V})\|_{rt}$ is non-empty. There exist tuples in $\|\pi_{\mathbf{G}_f}(\mathbf{V})\|_{rt}$ with $\mathbf{G}_f = g_f$.

Case 2: $g_f \in \pi^F_{\mathbf{G}_f}(\mathbf{R})$ and $g_f \notin \pi^F_{\mathbf{G}_f}(\|\mathbf{R}\|_{rt})$
$\sigma^F_{\mathbf{G}_f=g_f}(\|\mathbf{R}\|_{rt})$ is empty and thus, $\|\sigma_{\mathbf{G}_f=g_f}(\mathbf{V})\|_{rt}$ is empty. There do not exist tuples in $\|\pi_{\mathbf{G}_f}(\mathbf{V})\|_{rt}$ with $\mathbf{G}_f = g_f$.

Summarizing, the equality $\|\pi_{\mathbf{G}_f}(\mathbf{V})\|_{rt} = \pi^F_{\mathbf{G}_f}({}_{\mathbf{G_f},\mathbf{G}_o}\vartheta^F_\phi(\|\mathbf{R}\|_{rt}))$ holds.

Step 2: Same values of the ongoing grouping attributes $\mathbf{G}_o$ *for a fixed grouping attribute value* g_f

$$\begin{aligned}
& \|\pi_{\mathbf{G}_o}(\sigma_{\mathbf{G}_f=g_f}(\mathbf{V}))\|_{rt} = \pi^F_{\mathbf{G}_o}(\sigma^F_{\mathbf{G}_f=g_f}({}_{\mathbf{G_f},\mathbf{G}_o}\vartheta^F_\phi(\|\mathbf{R}\|_{rt}))) \\
\Leftrightarrow \quad & \|\pi_{\mathbf{G}_o}(\mathbf{O})\|_{rt} = \pi^F_{\mathbf{G}_o}(\sigma^F_{\mathbf{G}_f=g_f}(\|\mathbf{R}\|_{rt})) \\
\Leftrightarrow \quad & \|\pi_{\mathbf{G}_o}(\mathbf{X})\|_{rt} = \pi^F_{\mathbf{G}_o}(\|\sigma_{\mathbf{G}_f=g_f}(\mathbf{R})\|_{rt}) \\
\Leftrightarrow \quad & \|\pi_{\mathbf{G}_o}(\sigma_{\mathbf{G}_f=g_f}(\mathbf{R}))\|_{rt} = \|\pi_{\mathbf{G}_o}(\sigma_{\mathbf{G}_f=g_f}(\mathbf{R}))\|_{rt}
\end{aligned}$$

The ongoing grouping attributes values in relation $\sigma_{\mathbf{G}_f=g_f}(\mathbf{V})$ result from ongoing relation $\mathbf{O}$. The grouping attribute values in ${}_{\mathbf{G_f},\mathbf{G}_o}\vartheta^F_\phi(\|\mathbf{R}\|_{rt})$ result from $\|\mathbf{R}\|_{rt}$ (cf. Section 4.3). The ongoing selection σ and ongoing projection π remain valid as time passes by [MB20b] and $\|g_f\|_{rt} = g_f$.

Step 3: Same aggregate values for grouping attribute values (g_f, g_o).

$$\|\sigma_{\mathbf{G}_f=g_f \wedge \mathbf{G}_o=g_o}(\mathbf{V})\|_{rt} \stackrel{?}{=} \sigma^F_{\mathbf{G}_f=g_f \wedge \mathbf{G}_o=\|g_o\|_{rt}}({}_{\mathbf{G_f},\mathbf{G}_o}\vartheta^F_\phi(\|\mathbf{R}\|_{rt}))$$

The grouping attribute values are equal. We need to show the equality of the aggregate values v_i ($i \in [1,k]$):

$$\|\phi_i(\sigma^F_{\mathbf{G}_o=g_o \wedge RT=RT_o}(\mathbf{O}))\|_{rt} \stackrel{?}{=} \phi_i(\sigma^F_{\mathbf{G}_f=g_f \wedge \mathbf{G}_o=\|g_o\|_{rt}}(\|\mathbf{R}\|_{rt}))$$

Since the $\mathbf{G}_o$ values for a fixed grouping attribute value g_f are the same as in the instantiated case at a reference time (cf. step 2), we can choose $(g_o, RT_o) \in \pi_{\mathbf{G}_o}(\mathbf{O})$ with $rt \in RT_o$.

$$\begin{aligned}
& \|\sigma^F_{\mathbf{G}_o=g_o \wedge RT=RT_o}(\mathbf{O})\|_{rt} = \sigma^F_{\mathbf{G}_o=\|g_o\|_{rt}}(\sigma^F_{\mathbf{G}_f=g_f}(\|\mathbf{R}\|_{rt})) \\
\Leftrightarrow & \|\sigma^F_{\mathbf{G}_o=g_o \wedge RT=RT_o}(\mathbf{O})\|_{rt} = \sigma^F_{\mathbf{G}_o=\|g_o\|_{rt}}(\|\sigma_{\mathbf{G}_f=g_f}(\mathbf{R})\|_{rt}) \\
\Leftrightarrow & \|\sigma^F_{\mathbf{G}_o=g_o \wedge RT=RT_o}(\mathbf{O})\|_{rt} = \sigma^F_{\mathbf{G}_o=\|g_o\|_{rt}}(\|\mathbf{O}\|_{rt})
\end{aligned}$$

The equality holds since the following two properties hold.

Property 1: For any two tuples $o_1, o_2 \in \mathbf{O}$, if $rt \in o_1.RT$ and $rt \in o_2.RT$, the RT values of the tuples o_1 and o_2 are syntactically equal, i.e., $o_1.RT = o_2.RT$.

The set $\mathbf{Z}$ includes for all tuples $x_1, x_2 \in \mathbf{X}$ the RT value when tuples x_1 and x_2 are equal according to $\mathbf{G}_o$ and belong to the relation (cf. Figure 4.6). Then, function *splitRT* in Definition 9 guarantees that for the reference time of two tuples $x_3, x_4 \in \mathbf{X}$ and set $\mathbf{Z}$ any two split RT values of $splitRT(x_3.RT, \mathbf{Z})$ and $splitRT(x_4.RT, \mathbf{Z})$ are either equal or disjoint. These split RT values become the RT values of the tuples in $\mathbf{O}$. Thus,

$$\forall o_1, o_2 \in \mathbf{O}((rt \in o_1.RT \wedge rt \in o_2.RT) \Rightarrow o_1.RT = o_2.RT)$$

Property 2: If the $\mathbf{G}_o$ values of any two tuples in $\mathbf{O}$ are semantically equal at a reference time, the $\mathbf{G}_o$ values are also syntactically equal:

$$\begin{aligned}\forall o_1, o_2 \in \mathbf{O}((&rt \in o_1.RT \wedge rt \in o_2.RT \wedge \|o_1.\mathbf{G}_o\|_{rt} = \|o_2.\mathbf{G}_o\|_{rt}) \\ &\Rightarrow o_1.\mathbf{G}_o = o_2.\mathbf{G}_o)\end{aligned}$$

Since property 1 holds, the values of the ongoing grouping attributes $\mathbf{G}_o$ are canonicalized over the same set RT_x of reference times, i.e., $o_1.\mathbf{G}_o = \text{canon}(x_1.\mathbf{G_o}, RT_x)$ and $o_2.\mathbf{G}_o = \text{canon}(x_2.\mathbf{G_o}, RT_x)$. Since $\text{canon}(x_1.\mathbf{G_o}, RT_x) = \text{canon}(x_2.\mathbf{G_o}, RT_x)$, $o_1.\mathbf{G}_o = o_2.\mathbf{G}_o$.

Aggregate function ϕ_i is applied to the same set of input tuples and its result is calculated on fixed attribute values only, which are independent of the reference time rt. The aggregate values for group $(g_f, \|g_o\|_{rt})$ are the same in the ongoing relation $\mathbf{V}$ and the expected result relation at reference time rt.

Summarizing, at each reference time, the ongoing relation in Theorem 3 consists of the same tuples as the expected result relation at that reference time. □

Example 18. *Consider the aggregation* ${}_{PID,VT}\vartheta_{count(*)}(\mathbf{P})$ *of our running example in Example 14 with fixed grouping attribute* $\mathbf{G}_f = (PID)$*, ongoing grouping attribute* $\mathbf{G}_o = (VT)$*, and aggregate function* $\phi = [count(*)]$*. Figure 4.3 illustrates the equivalence in Theorem 3 for aggregation* ${}_{PID,VT}\vartheta_{count(*)}(\mathbf{P})$*. The input tuples and the result tuples are shown with the same color-coding as in Figure 4.2.*

In Figure 4.3, part 1 corresponds to determining the fixed groups in line 2 in the equivalence; part 2 corresponds to determining the ongoing groups $\mathbf{O}$ *in lines 3 - 7 in the equivalence; and*

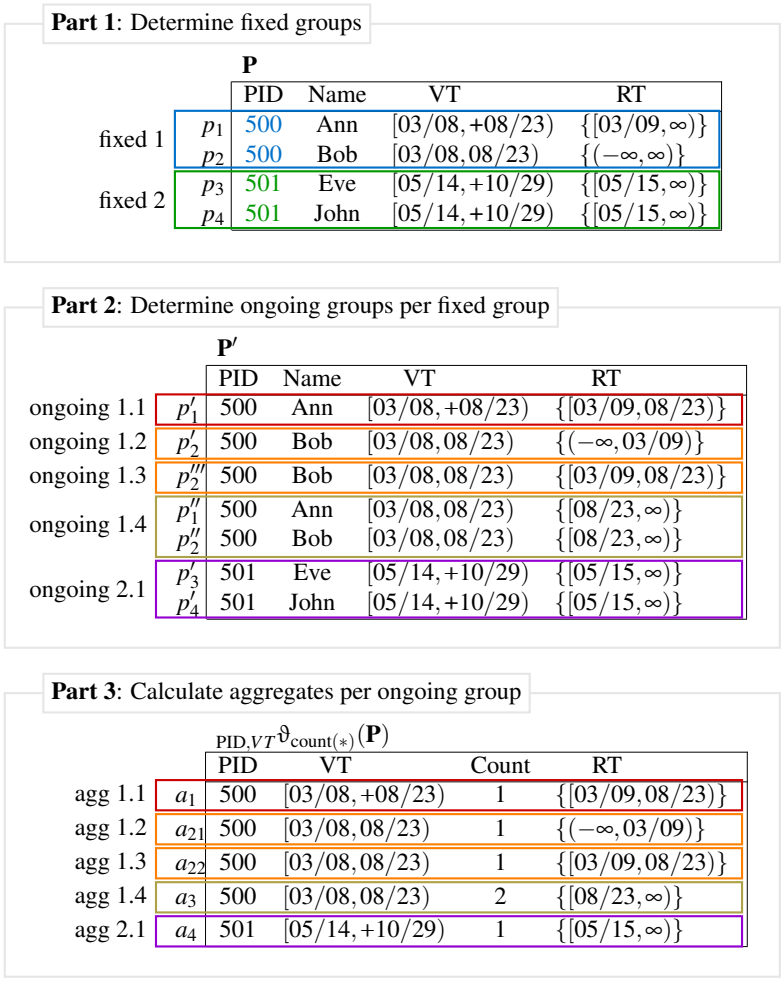


Figure 4.3: Illustration of the aggregation operator.

part 3 corresponds to calculating the aggregate values in the last two lines in the equivalence and to the returned tuples in line 2 in the equivalence.

The projection in line 2 determines the values of the fixed grouping attributes, $\{(500),(501)\}$. *One of the fixed groups is* $g_f = (500)$. *The selection in line 2 determines all tuples that belong to fixed group* $g_f = (500)$*:* $\mathbf{X} = \{p_1, p_2\}$.

In lines 3 - 7, the ongoing groups in fixed group $g_f = (500)$ *are determined. The set of RT values,* $\mathbf{Z} = \{p_1.RT, p_2.RT, \{[08/23,\infty)\}\}$, *contains for any two tuples* $x_1, x_2 \in \mathbf{X}$ *the reference times when they are equal according to the ongoing grouping attribute* $\mathbf{G}_o = \{VT\}$ *and both tuples belong to the relation,* $x_1.RT \wedge x_2.RT$. *For tuple* $p_1 \in \mathbf{X}$, *the split RT values are calculated with respect to* $\mathbf{Z}$ *(line 7). The result of* splitRT *is* $\{\{[03/09,08/23)\},\{[08/23,\infty)\}\}$ *as discussed in Example 17. For each reference time* $RT \in splitRT(\cdot)$, *the value of* $\mathbf{G}_o = \{VT\}$ *is canonicalized*

according to RT. For tuple p_1, this results in the tuples p'_1 and p''_1 in Figure 4.3. The same procedure is done for the second tuple in **X***, tuple p_2. Summarizing,* **O** *consists for fixed grouping attribute value $g_f = (500)$ of the tuples:* $\mathbf{O} = \{p'_1, p''_1, p'_2, p''_2, p'''_2\}$.

In the last two lines of the equivalence, the projection retrieves each ongoing group in **G** *and the aggregate functions ϕ are calculated on each ongoing group. In our example,* **O** *consists of four ongoing groups for fixed group $g_f = (500)$:*

- *ongoing 1.1:* $\{p'_1\}$
- *ongoing 1.2:* $\{p'_2\}$
- *ongoing 1.3:* $\{p'''_2\}$
- *ongoing 1.4:* $\{p''_1, p''_2\}$

The count for ongoing groups 1.1, 1.2, and 1.3 is one and the count for ongoing group 1.4 is two. The tuples returned for fixed group $g_f = (500)$ are tuples a_1, a_{21}, a_{22}, and a_3.

The same procedure is executed for the second fixed group $g_f = (501)$ and tuple a_4 is returned for this fixed group.

Summarizing, the aggregation result consists of the five tuples a_1, a_{21}, a_{22}, a_3, and a_4. This result is equivalent to the result given in Figure 4.3. Tuples a_{21} and a22 are merged into a single tuple a_2 in Figure 4.3.

4.5 Implementation

This section describes and analyzes our implementation of the aggregation operator for ongoing relations in the PostgreSQL database system. We extended all components in the query processing pipeline: parser and parse tree, analyzer and query tree, optimizer and plan tree, and executor and execution tree. The key idea of the execution algorithm is to first group tuples according to the fixed grouping attributes, then divide each fixed group further into ongoing groups according to the ongoing grouping attributes and the *RT* attribute, and finally, calculate the aggregate functions for each ongoing group.

4.5.1 Extensions to Parser, Analyzer, and Optimizer

We extended the syntax of an SQL query with the following grouping clause:

```
groupOngoing by fixed(expr_list) \
                ongoing(expr_list) rt (expr_list)
```

The `expr_lists` refer to lists of attributes from the input relation. The first `expr_list` includes the fixed grouping attributes, the second `expr_list` includes the ongoing grouping attributes, and the last `expr_list` consists of the reference time attribute. The aggregation ${}_{\text{PID},VT}\vartheta_{\text{count}(*)}(\mathbf{P})$ in Example 14 is then expressed as follows in SQL:

```
SELECT PID, VT, count(*), RT
FROM P
GROUPONGOING BY FIXED(PID) ONGOING(VT) RT(RT);
```

We extended the SQL syntax with the `groupOngoing` clause to illustrate and evaluate our aggregation for ongoing relations. Instead of extending the SQL syntax, a database system could translate the usual `group by` clause into our `groupOngoing` clause by automatically determining the fixed and ongoing grouping attributes and the *RT* attribute and then, internally, proceed with the parse, query, and execution tree discussed in the remainder of this section.

We extend the aggregation node of the parse tree and the query tree with a grouping structure that distinguishes between the three different grouping attributes:

```
typedef struct GroupOngoingClause
{
  NodeTag   type;
  List      *fixedAttributes;
  List      *ongoingAttributes;
  List      *rtAttribute;
} GroupOngoingClause;
```

We distinguish between the fixed grouping attributes, the ongoing grouping attributes, and reference time attributes since we apply the existing grouping mechanisms to the fixed grouping

attributes and add an ongoing-specific grouping mechanism for the other grouping attributes in the executor.

For the ongoing aggregation, we use sorted aggregation. This means that the optimizer chooses a plan that sorts the input relation according to the fixed grouping attributes first and then applies the ongoing aggregation node to the sorted input. This is illustrated in Figure 4.4. Although currently not implemented, our approach is also applicable to hashed aggregation, the alternative aggregation method offered by PostgreSQL. The aggregation method dictates how the fixed groups are determined; our approach for determining the ongoing groups within a fixed group in the executor (cf. next section) can be used independently of the aggregation method.

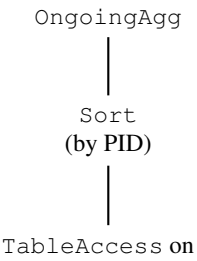


Figure 4.4: Plan tree for ${}_{\text{PID},VT}\vartheta_{\text{count}(*)}(\mathbf{P})$.

4.5.2 Execution Algorithm

The execution of an executor node is divided into three phases: `ExecInit<Node>`, `Exec<Node>`, and `ExecEnd<Node>`. The first phase is for the initialization, the second phase for the execution, and the last phase for the finalization of the evaluation algorithm. For the ongoing aggregation, the node is called `OngoingAgg`.

The implementation of the aggregation operator on ongoing relations is a two step process: (1) We sort the input relation according to the fixed grouping attributes; the tuples that belong to the same fixed group are grouped together and (2) our evaluation algorithm then divides each fixed group further into sub-ongoing groups according to the ongoing grouping attributes and the *RT* attribute and calculates the aggregate values for each ongoing group.

For a fixed group, the ongoing groups and their aggregate values are calculated as illustrated in Figure 4.5 and discussed below. An ongoing group is represented with (1) a single master tuple whose fixed and ongoing attribute values are representative for the tuples in the group and whose *RT* value is the group's reference time *RT* and with (2) the aggregate values that are

incrementally calculated. Ongoing groups are incrementally built by manipulating the group's reference time RT. The evaluation algorithm starts with an ongoing group that is represented by the first tuple in the fixed group. Each subsequent tuple t_i in the fixed group then splits an ongoing group into two ongoing groups: the first ongoing group g_1 that consists of the reference times when the group and the tuple are equal according to the ongoing grouping attributes and have common reference times RT and the second ongoing group g_2 whose reference time is the reference time of the original ongoing group minus the reference time of the first split ongoing group. A new ongoing group g_3 with tuple t_i as its master tuple is created for the reference times when the tuple is not equal to any ongoing group. These reference times are included in the group's RT value. Whenever a tuple is conceptually added to an ongoing group (groups g_1 and g_3), the aggregate values of the group are updated.

Our evaluation algorithm supports pipelining such that the tuples that belong to a fixed group and the tuples that belong to an ongoing group do not need to be materialized. To make this possible, we sort the input relation according to the fixed grouping attributes and incrementally calculate the aggregate values. Sorting ensures that tuples with the same fixed grouping attribute values can be fetched consecutively and the evaluation algorithm can process all tuples of a fixed group and produce its result tuples before starting the next fixed group. The incremental calculation of the aggregate values ensures that the evaluation algorithm does not need to materialize the tuples that belong to an ongoing group but, instead, can update the aggregate values of an ongoing group whenever a tuple that belongs to this group is processed.

The evaluation algorithm of the aggregation operator for ongoing relations in Algorithm 7 is implemented in PostgreSQL as the executor function `ExecOngoingAgg`. The function is integrated into the pipelining architecture of PostgreSQL and at each invocation either a single result tuple is returned, or ω to indicate the end of the operation. The input is a context node, n, that keeps variables between different invocations:

- `subnode`: stores a reference to its input; the tuples fetched from the subnode in the execution tree are sorted according to the fixed grouping attributes.

- `out`: stores an output tuple.

- `fixedGroupTuple`: stores a tuple of the currently processed fixed group; if the end of the current fixed group is reached, the variable stores the first tuple of the next fixed group.

- `sameFixedGroup`: boolean that is *true* while tuples of the same fixed group are processed; it is also *true* to indicate that the next fixed group should be processed.

- `ongoingGroups`: stores a list of the ongoing groups for a fixed group. An ongoing group consists of the master tuple, i.e., the tuple that represents the group, and the aggregate state of each aggregate function for this group (e.g., the average function keeps a sum and a count as aggregate state).

We use the built-in mechanisms of PostgreSQL to manage the aggregate states and to produce an output tuple. Function *createAndInitGroup* sets the master tuple of the group and uses the built-in functions to initialize the aggregate state of each aggregate function. Function *advanceAggregates* updates the aggregate state of each aggregate function of a group with the attribute values of a given tuple. Function *finalizeAggregate* calculates the result aggregate values for each aggregate function based on the aggregate state. Function *assembleOutputTuple* creates the output tuple with the correct schema; it uses the attribute values of the master tuple for the non-aggregate attributes of an ongoing group and the finalized aggregate values for the aggregate functions.

Figure 4.5 illustrates the invocations of `ExecOngoingAgg`, which produce the result tuples for the fixed group *PID = 500* of the aggregation ${}_{\text{PID},VT}\vartheta_{\text{count}(*)}(\mathbf{P})$ of our running example in Example 14. Before the first invocation of `ExecOngoingAgg`, the input relation was sorted according to the fixed grouping attribute, *PID*, and the subnode will returns the tuples of $\mathbf{P} = \{p_1, p_2, p_3, p_4\}$ in this order. Note that tuples p_1 and p_2 belong to the fixed group *PID = 500* and tuples p_3 and p_4 belong to the fixed group *PID = 501*.

In the following, we will discuss the first four invocations of `ExecOngoingAgg`, which will produce all result tuples for fixed group *PID = 500*, a_1, a_2, and a_3, in the first three invocations and will start processing the tuples of the next fixed group *PID = 501* in the forth invocation. In the first invocation of `ExecOngoingAgg`, `fixedGroupTuple` = `curr` are set to p_1 and `sameFixedGroup` = *true*. Since `ongoingGroups` is empty (line 9) and `curr`.RT is non-empty (line 22) a new ongoing group is created with p_1 as its master tuple and its count aggregate value set to 1. The next tuple, p_2, is fetched into `curr` (line 27). Tuple p_2 belongs to the same fixed group as p_1 (line 28) and the while-loop continues. The `commonRT` of p_2 and ongoing group (masterTuple=p_1, masterTuple.RT= $\{[03/09,\infty)\}$, count=1) is $\{[08/23,\infty)\}$. The ongoing group is split into two ongoing groups (line 13): tuple p_2 matches the copied group, its *RT* is set to `commonRT`, and its count is increased to 2; the *RT* of the original ongoing group is updated to $\{[03/09,08/23)\}$. The *RT* of p_2 is set to $\{(-\infty,08/23)\}$ (line 20). Since *curr.RT* is non-

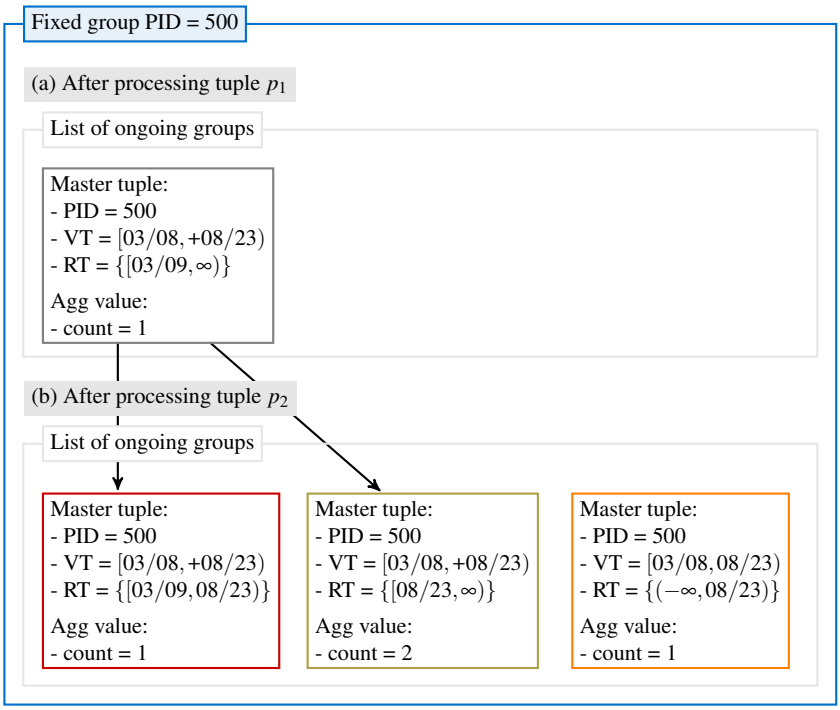


Figure 4.5: Evaluation algorithm for fixed group *PID = 500*.

empty, a new group is created and appended to the list of ongoing groups. The list of ongoing groups contains the three groups shown in Figure 4.5. The next tuple of subnode, p_3, is fetched into `curr`. This tuple does not belong to the same fixed group (line 28), the tuple is saved in `fixedGroupTuple` and `sameFixedGroup` is set to *false*. The while loop terminates and since the list of ongoing groups is non-empty, output tuple $a_1 = (500, [03/08, +08/23), 1, \{[03/09, 08/23)\})$ is produced (line 33). In the second invocation, `sameFixedGroup` is *false*, the while loop is not entered, and the list of ongoing groups is non-empty. Output tuple $a_3 = (500, [03/08, +08/23), 2, \{[08/23, \infty)\})$ is produced. In the third invocation, `sameFixedGroup` is *false*, the while loop is not entered, and the list of ongoing groups is non-empty. Output tuple $a_2 = (500, [03/08, 08/23), 1, \{(-\infty, 08/23)\})$ is produced. The list of ongoing groups is empty and `sameFixedGroup` is set to *true*. In the forth invocation, the next fixed group is processed: `sameFixedGroup` is *true* and `fixedGroupTuple` = `curr` = p_3.

```
Procedure: ExecOngoingAgg(n)
Input: Node n in execution tree.
Output: A single output tuple or ω.
1  Copy variables of n to local;
2  if first call then
3  |  fixedGroupTuple ← next tuple from subnode;
4  |  sameFixedGroup ← true;
5  |  ongoingGroups ← [ ];
6  end
7  curr ← fixedGroupTuple;
8  while sameFixedGroup ∧ curr ≠ ω do
9  |  foreach group ∈ ongoingGroups do
10 |  |  commonRT ← (group.masterTuple.RT ∧ curr.RT ∧ (group.masterTuple.G_o =
   |  |   curr.G_o));
11 |  |  if group.masterTuple.RT = commonRT then
12 |  |  |  advanceAggregate(group, curr);
13 |  |  else if commonRT ≠ ∅ then
14 |  |  |  copiedGroup ← copy(group) ;
15 |  |  |  copiedGroup.masterTuple.RT ← commonRT;
16 |  |  |  advanceAggregate(copiedGroup, curr);
17 |  |  |  ongoingGroups.append(copiedGroup);
18 |  |  |  group.masterTuple.RT ← group.masterTuple.RT ∧¬ commonRT;
19 |  |  end
20 |  |  curr.RT ← curr.RT ∧¬ commonRT;
21 |  end
22 |  if curr.RT ≠ ∅ then
   |  |  // Sets master tuple and initializes the agg function states.
23 |  |  newGroup ← createAndInitGroup(masterTuple=curr);
24 |  |  advanceAggregates(newGroup, curr);
25 |  |  ongoingGroups.append(newGroup);
26 |  end
27 |  curr ← next tuple from subnode;
28 |  if fixedGroupTuple.G_f ≠ curr.G_f then
29 |  |  fixedGroupTuple ← curr;
30 |  |  sameFixedGroup ← false;
31 |  end
32 end
33 if ongoingGroups is not empty then
34 |  group ← ongoingGroups.first();
35 |  finalizeAggregates(group);
36 |  out ← assembleOutputTuple(group);
37 |  sameFixedGroup ← (is ongoingGroups empty?);
38 else
39 |  out ← ω;
40 end
41 Copy local variables to node n;
42 return out;
```

Algorithm 7: Executor function.

4.5.3 Analysis

Complexity Analysis The runtime complexity of the evaluation algorithm scales quadratically with the result size complexity in the worst case. The reason is that for each tuple in a fixed group, the evaluation algorithm iterates through each ongoing group produced so far (cf. line 9 in Algorithm 7). However, the empirical evaluation in Section 4.6.3 shows that the runtime complexity scales linearly with the result size in a wide variety of practical cases.

Property 1. *The number of ongoing groups in a fixed group is quadratic in the number of tuples n in the fixed group, i.e.,*

$$\#\ \textit{ongoing groups per fixed group} = \mathcal{O}(n^2)$$

The number of ongoing groups in a fixed group depends on the reference times when the tuples are equal according to the ongoing grouping attributes $\mathbf{G}_o$ and belong to the relation, i.e., on $RT_{ij} = (r_i.\mathbf{G}_o = r_j.\mathbf{G}_o \wedge r_i.RT \wedge r_j.RT)$ for tuples r_i and r_j of the fixed group (cf. line 10 in Algorithm 7). Figure 4.6 illustrates the RT_{ij} values for a fixed group with tuples $\{r_1, r_2, r_3, r_4, r_5\}$.

A new ongoing group is created when the set of tuples that belong to an ongoing group at reference time rt changes at reference time $rt+1$. This happens when at least one of the following four events occur:

(1) a tuple r_p does *not belong* to the relation at reference time rt ($rt \notin r_p.RT$) and it *does belong* to the relation at $rt+1$ ($rt+1 \in r_p.RT$) and *is equal* to the tuples in the ongoing group at reference time $rt+1$,

(2) a tuple r_p *belongs* to the relation at reference time rt ($rt \in r_p.RT$) and *is equal* to the tuples in the ongoing group at reference time rt and it *does not belong* to the relation at reference time $rt+1$ ($rt+1 \notin r_p.RT$), and

(3) a tuple r_p *belongs* to the relation at reference time rt ($rt \in r_p.RT$) and *is equal* to the tuples in the ongoing group at reference time rt and it *belongs* to the relation at $rt+1$ ($rt+1 \in r_p.RT$) but is *not equal* to the tuples in the ongoing group at reference time $rt+1$.

(4) a tuple r_p *belongs* to the relation at reference time rt ($rt \in r_p.RT$) and is *not equal* to the tuples in the ongoing group at reference time rt and it *belongs* to the relation at $rt+1$ ($rt+1 \in r_p.RT$) but *is equal* to the tuples in the ongoing group at reference time $rt+1$.

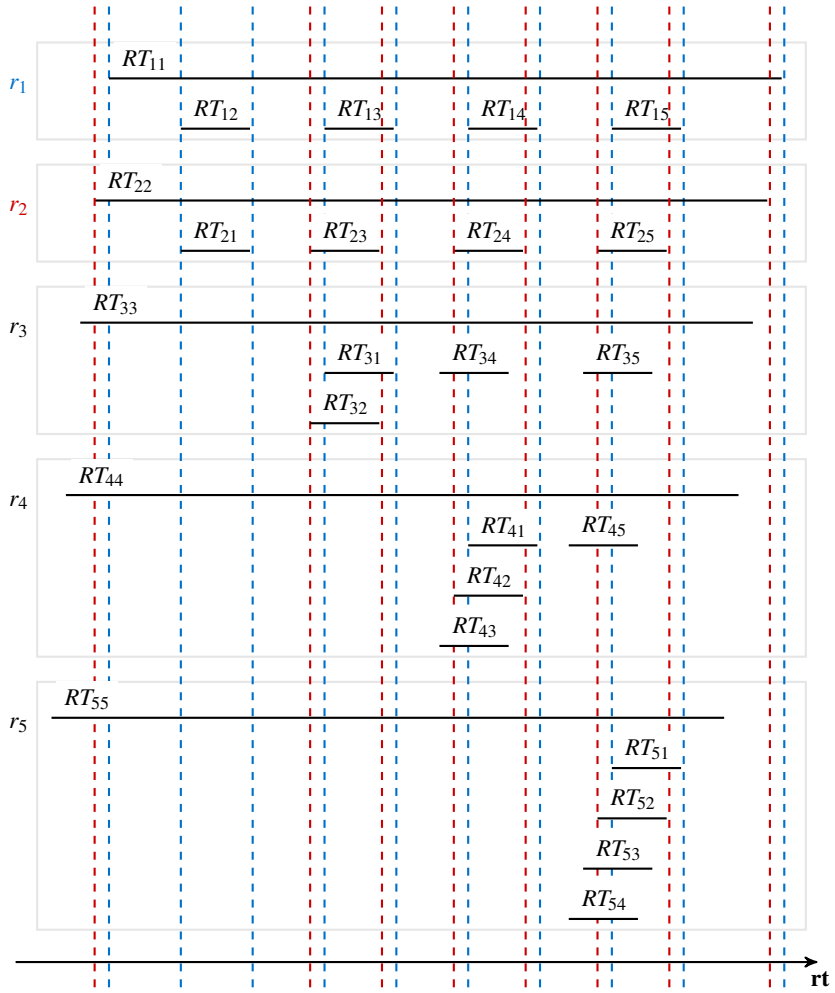


Figure 4.6: The split reference times that the tuples in a fixed group create ($RT_{ij} = (r_i.\mathbf{G}_o = r_j.\mathbf{G}_o \wedge r_i.RT \wedge r_j.RT)$).

These events correspond to the boundaries, i.e., start and end points, of the intervals in RT_{ij}. Recall that we represent the value of the RT attribute and the result of predicates on ongoing values as a list of maximal, non-overlapping fixed intervals. We refer to these boundaries as *split reference times*. For instance, the split reference times for tuple r_1 in Figure 4.6 are the blue dotted vertical lines at the start and end of RT_{11}, RT_{12}, RT_{13}, RT_{14}, and RT_{15}. As examples, the start point of RT_{11} describes the first event, the end point of RT_{11} describes the second event, the end point of RT_{12} describes the third event, and the start point of RT_{12} describes the fourth event.

We determine an upper bound for the number of split reference times and thus, the number of ongoing groups by calculating the pair-wise equality of the tuples, $RT_{ij} = (r_i.\mathbf{G}_o = r_j.\mathbf{G}_o \wedge r_i.RT \wedge r_j.RT)$, between all tuples in a fixed group (cross product).

For each tuple pair r_i and r_j and ongoing grouping attributes $\mathbf{G}_o = \{O_1, \ldots, O_m\}$, there exist at most $2 \times |RT_{ij}|$ many split reference times. The cardinality $|RT_{ij}|$ is the number of fixed intervals that is used to represent the RT_{ij} value. It is twice the cardinality since both, the start and end point of an RT interval, can lead to different ongoing groups. As an example, RT_{12} in Figure 4.6 is represented with a single fixed interval, resulting in two split reference times. Two tuples belong to the same ongoing group only if all their ongoing grouping attribute values are equal and both tuples belong to the relation. The cardinality of the conjunction is bound by the sum of its input cardinalities [MB20c]. Then,

$$2 \times |RT_{ij}| \leq 2 \times (\sum_{k=1}^{m} (|r_i.O_k = r_j.O_k|) + |r_i.RT| + |r_j.RT|)$$

Each tuple is compared to n tuples in the fixed group and the number of split reference times per tuple is bound by

$$\sum_{j=1}^{n} (2 \times |RT_{ij}|)$$

As an example, the number of split reference times of tuple r_1 in Figure 4.6 is 10: the number of tuples in the fixed group is five, $n = 5$, and each RT_{1j} is represented with a single fixed interval: $\sum_{j=1}^{5}(2 \times 1) = 10$.

For all n tuples in a fixed group, the upper bound of ongoing groups is n times the upper bound for one tuple:

$$\begin{aligned}
&\sum_{i=1}^{n} \sum_{j=1}^{n} (2 \times |RT_{ij}|) \\
&\leq \sum_{i=1}^{n} \sum_{j=1}^{n} (2 \times \sum_{k=1}^{m} (|r_i.O_k = r_j.O_k|) + |r_i.RT| + |r_j.RT|)
\end{aligned}$$

The cardinality of the result of the equality of ongoing time intervals (the data type of the valid time attribute) and the cardinality of the RT attributes are independent of the number of tuples. Then, $|RT_{ij}|$ is also independent of the number of tuples. Thus, the number of ongoing groups for a fixed group is

$$n * \mathcal{O}(n) = \mathcal{O}(n^2)$$

Property 2. *The total number of result tuples is the number of fixed groups times the number of ongoing groups per fixed groups, i.e.,*

$$\#\,\textit{result tuples} = \#\,\textit{fixed groups} \times \mathcal{O}(n^2)$$

For any given relation, an aggregation resulting in larger fixed group sizes conversely implies a smaller number of fixed groups by the same factor. Thus, the effective complexity with respect to the fixed group size is linear instead of quadratic. We observe this effect in our empirical evaluation in Figure 4.10.

Storage Requirements The memory storage required by the evaluation algorithm is limited to the storage requirements for a single fixed group. The reason is that the evaluation algorithm first processes the tuples of a fixed group before processing the tuples of the next fixed group. Only after all tuples of a fixed group have been processed, the ongoing groups in this fixed group can be produced. Thus, the evaluation algorithm needs to store all ongoing groups of a single fixed group in memory. After the ongoing groups have been produced, the memory context for the fixed group can be cleared, i.e., the memory allocated while processing the fixed group is freed.

As discussed in the complexity analysis, there exist at most $\mathcal{O}(n^2)$ ongoing groups per fixed group (n is the number of tuples in a fixed group). The worst case memory consumption for the ongoing groups is

$$\mathcal{O}(n^2) \times \textit{size}(\text{ongoing group metadata})$$

The size of an ongoing group consists of the size of the master tuple, which is equal to the size of an input tuple, and the sizes of the intermediate aggregates for the used aggregate functions:

$$\begin{aligned} &\textit{size}(\text{ongoing group metadata}) \\ &= \textit{size}(\text{master tuple}) + \sum_{1}^{\#\text{aggregates}} \textit{size}(\text{intermediate aggregate}) \end{aligned}$$

The size of most intermediate aggregate values is equal to the size of an integer (e.g., for count, min, max, sum). The *avg* aggregate function requires two integers since it stores a sum and a count as intermediate aggregate values.

Additionally to the ongoing groups, the evaluation algorithm allocates memory to compute the intermediate RT values (lines 10 and 20 in Algorithm 7).

4.6 Evaluation

This section compares runtime and result size of our ongoing aggregation with the standard aggregation in database systems, which is only applicable to fixed grouping attributes and relations without a reference time attribute. A non-trivial reference time and equal ongoing intervals have an impact on the performance of the ongoing aggregation and the number of ongoing groups in the result. To evaluate the scalability of our approach, we vary the sizes of the fixed groups and the number of aggregate functions in the aggregation query.

4.6.1 Setup

The empirical evaluation is conducted on a 3.40 GHz machine with 16GB main memory and an SSD. The client and the database server run on the same machine. We use the PostgreSQL 9.4.0 kernel extended with our implementation of the aggregation operator on ongoing relations.

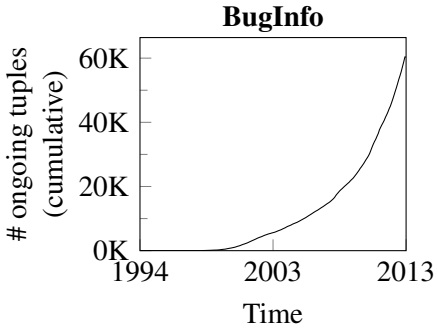
Table 4.1 summarizes the real-world and synthetic data sets. As real-world data set, we use *MozillaBugs* [LPD13] that records the history of bugs in the Mozilla project. The *BugInfo* table records general information about a bug: ID, product, component, operating system, severity, and valid time. A bug has a trivial reference time, i.e., $RT = \{(-\infty, \infty)\}$. Bugs that have not been resolved as of the date of the data export have ongoing valid time intervals. For the valid time, we use a granularity of years to avoid the trivial case in which most ongoing groups are based on a single tuple.

Table 4.1: Characteristics of the experiment data sets.

	MozillaBugs	$\mathbf{D^{group}}$	$\mathbf{D^{agg}}$	$\mathbf{D^{rt}}$	$\mathbf{D^{vt}}$
Cardinality	394,878	1M	1M	1M	1M
# ongoing	60,372 (15%)	20%	10%	20%	0% - 100%
# non-trivial RT	0%	0%	1%	0% - 100%	0%
Time intervals	$\{[a, now), [c, d)\}$			$\{[a, now), [c, d)\}$	
Fixed-group size	Figure 4.7b	10 - 100	100	100	100

Figure 4.7a shows the distribution of the start points of the ongoing intervals in *MozillaBugs*. 50% of the tuples with ongoing intervals in *BugInfo* are located within the last two years of the history. For experiments with an increasing number of tuples we grow the size of the real-world data set by growing the history of *BugInfo* backward. This means that the percentage of ongoing intervals decreases as the data size grows. Figure 4.7b shows the distribution of the size of the

fixed groups in *BugInfo*. Most tuples belong to fixed groups of small sizes, but there is a single extremely large fixed group with nearly 14k tuples.



(a) Start point distribution of ongoing intervals.

Size of fixed group	# tuples
0 - 1k	292k
1k - 2k	41k
2k - 3k	16k
3k - 4k	21k
4k - 5k	4k
5k - 6k	0
6k - 7k	7k
13k - 14k	14k

(b) Size distribution of the fixed groups.

Figure 4.7: Property distributions in *MozillaBugs*.

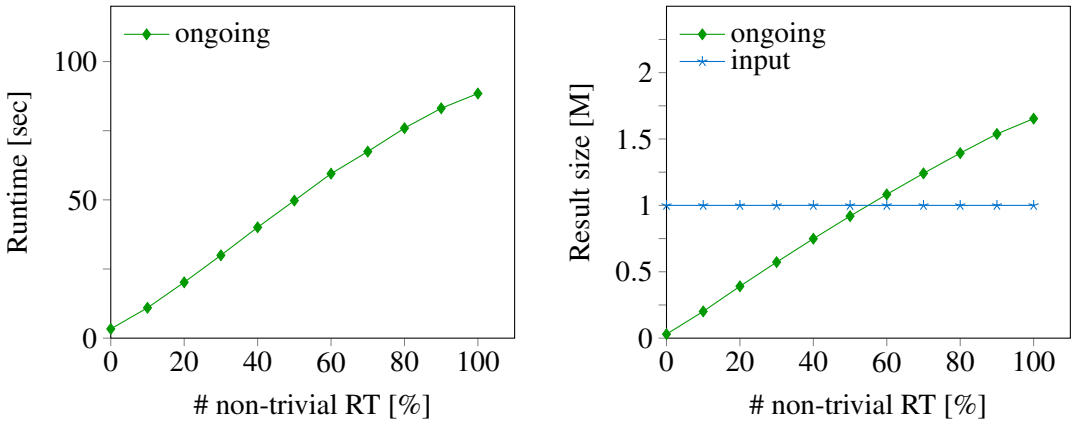
We compare our aggregation operator for ongoing relations with the standard aggregation operator in PostgreSQL, which is only applicable to fixed grouping attributes and relations without a reference time attribute. This allows us to evaluate the impact of additionally grouping according to ongoing attributes and the reference time in the ongoing aggregation operator. We refer to our aggregation as *ongoing* and to the standard database aggregation as *fixed*.

For the synthetic and the real-world data sets, we use the aggregation query $Q_{\phi}^{\vartheta}(\mathbf{R})$ on relation $\mathbf{R}$ with ϕ as the set of aggregate functions. Multiple aggregate functions (*mult*) include *avg*, *max*, *min*, *count*, and *sum*. For the synthetic data sets, we use a generated attribute as fixed grouping attribute and the valid time as ongoing grouping attribute. For the real-world data set, the aggregation query determines the aggregate functions over bugs that correspond to the same product, component, operating system, and have the same severity (fixed grouping attributes) and that are open at the same valid time (ongoing grouping attribute).

4.6.2 Ongoing Groups

The main factors that result in more ongoing groups and thus, have an impact on the performance of the ongoing approach are the presence of non-trivial reference time values and the percentage of ongoing intervals. Both tend to split an ongoing group into ongoing sub-groups (cf. Algorithm 7).

Non-trivial *RT*s We evaluate the effect of a non-trivial reference time *RT* on the runtime and result size of aggregation queries. Data set D^{rt} contains a varying percentage of non-trivial *RT* values, which are generated by first choosing a shared timepoint t_{sh}, and for each tuple generating a randomized start point 1 - 180 days earlier than t_{sh} and a randomized end point 1 - 180 days later than t_{sh}. This scheme avoids the uninteresting case where none of the *RT* values overlap, which in turn would make aggregation trivial as each ongoing group would only have one tuple to aggregate. Figure 4.8 shows the runtime and result size of the aggregation query on D^{rt}. The result size grows linearly with the percentage of non-trivial *RT* values (Figure 4.8b). This is due to each non-trivial *RT* value leading to the creation of a bounded number of additional ongoing groups within the fixed group due to its different start and end points. The runtime increase is linear as well (cf. Figure 4.8a), indicating that our implementation has no additional runtime complexity beyond the inherent constraint of requiring at least $O(1)$ time per result tuple. We also see that for high percentages of different, non-trivial *RT* values, the number of ongoing groups can surpass the number of original input tuples. However, this effect is unlikely to occur in practical usage, for two reasons: (1) original data sets usually consist of base tables whose tuples have a trivial *RT*, and (2) queries with ongoing predicates and functions usually lead to similar *RT* values in the result relation ([MB20b], [MB20a]). Thus, we would expect a very low percentage of different, non-trivial *RT* values as inputs to aggregation queries in practice.

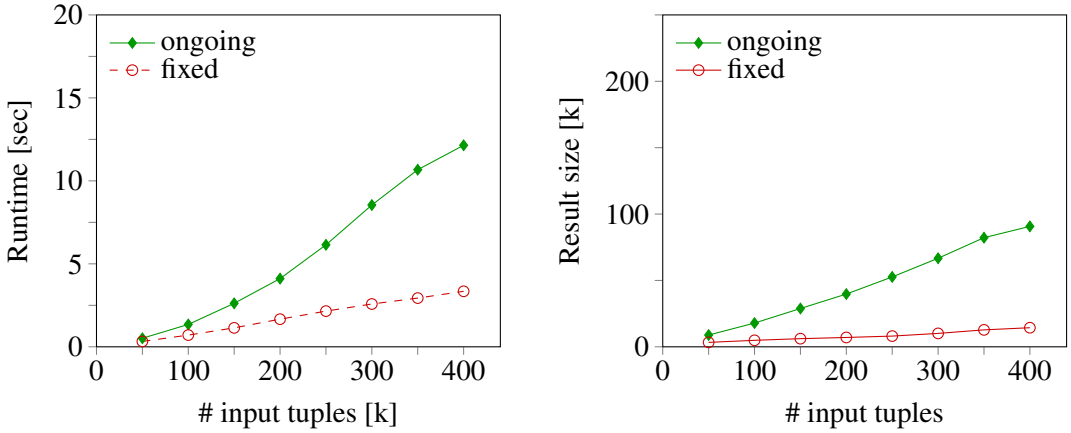


(a) Runtime. (b) Result size.

Figure 4.8: Percentage of non-trivial *RT* values ($Q^{\vartheta}_{\text{count}}(D^{rt})$).

Ongoing Percentage We use the *MozillaBugs* data set, which naturally has a lower percentage of ongoing tuples at larger sizes, to evaluate the effect that the different percentages of ongoing tuples have on the runtime and result size of aggregation queries. The results are shown in Figure 4.9. At a higher number of tuples, where ongoing tuples are less common, the growth in

both runtime (Figure 4.9a) and result size (Figure 4.9b) is sublinear. This is because the ongoing tuples are equal to a fixed tuple only at at most one reference time, which leads to a higher number of ongoing groups being needed to represent the overall result. The consistently linear growth of the fixed result size shows that the fixed grouping does not cause this effect.



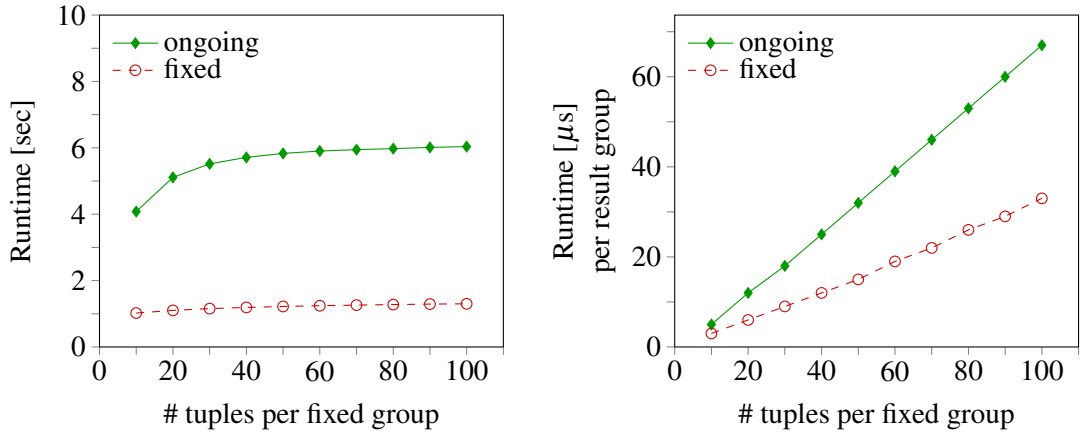
(a) Runtime. (b) Result size.

Figure 4.9: Percentage of ongoing intervals ($Q^{\vartheta}_{\text{count}}(\mathbf{B})$).

4.6.3 Scalability

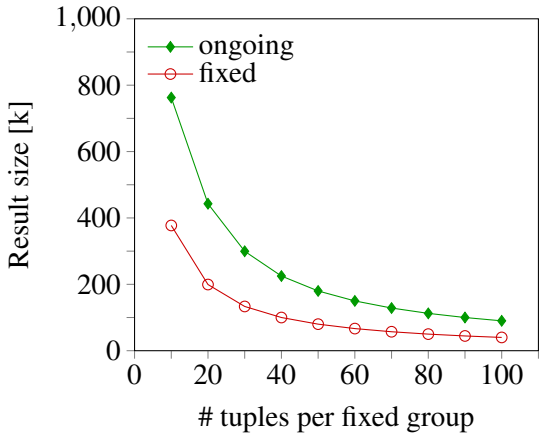
Fixed Group Size We evaluate the effect of the size of the fixed groups on the runtime of our approach. Data set D^{group} has a varying fixed group size, with the number of fixed groups varied accordingly, so that the overall size of the data set is constant at 1 million tuples. For the 20% ongoing tuples, we always use the same start dates to avoid creating more ongoing groups due to the ongoing valid time; this allows us to focus on the effects of varying the size of fixed groups. Section 4.6.2 evaluates the effects of increased splitting into ongoing groups within a fixed group for RT as well as for additional different ongoing VTs. Figure 4.10a shows the resulting runtimes and Figure 4.10c shows the resulting result sizes. For both metrics, our approach has a constant multiplicative overhead compared to the fixed approach. Figure 4.10b shows that each approach takes $O(\text{fixed group size})$ in order to calculate its result group (fixed approach) or multiple groups (ongoing approach).

Number of Aggregate Functions We evaluate the effect of additional aggregate functions within a query. For aggregation query $Q^{\vartheta}_{mult}(\mathbf{B})$, we cumulatively add the following aggregate functions, in order: avg, max, min, count, sum. Figure 4.11 shows that the effect of adding



(a) Runtime.

(b) Runtime ratio.



(c) Result size.

Figure 4.10: Size of the fixed groups ($Q^{\vartheta}_{\text{count}}(D^{group})$).

additional aggregate functions to an aggregation query is negligible. This confirms that, just as in the fixed approach, determining the groups and iterating over each group's input tuples is the primary component of the required runtime, not the actual evaluation of the aggregate function. All the grouping and RT adjustment logic is evaluated once, not once per aggregate function.

4.7 Conclusions

This paper proposes the first solution that evaluates the aggregation operator on ongoing relations to results that remain valid as time passes by. Our approach splits the *RT* values of the input tuples, such that we get groups of tuples with equal grouping attribute values and the same

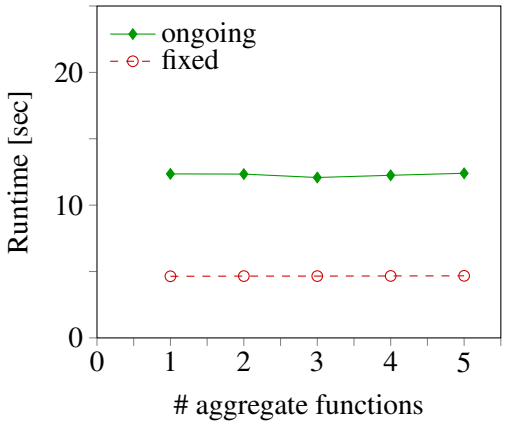


Figure 4.11: Number of aggregate functions ($Q^{\vartheta}_{mult}(\mathbf{B})$).

reference time RT. The aggregation result is an ongoing relation that includes a single, aggregated tuple for each group. We integrated our approach into the query processing pipeline of PostgreSQL and leverage the existing, optimized strategies of database systems to determine the fixed groups according to the fixed grouping attributes and to calculate the aggregate values. We intertwine the incremental calculation of the fixed and ongoing groups and the incremental calculation of the aggregate values to avoid materializing the tuples that belong to an aggregation group.

As future work, we want to add support for additional functions on ongoing data types like the duration function whose result are ongoing integers. In the presence of ongoing integers, we plan to extend our aggregation operator on ongoing relations to support aggregate functions on ongoing values as well. Depending on the representation of ongoing integers, this might require further adjustments of the reference time RT of an ongoing group.

CHAPTER 5

Conclusions and Future Work

In this thesis, we propose an approach that can efficiently store ongoing time points and time intervals in database systems and that evaluates queries with predicates and functions on ongoing attributes to results that remain valid as time passes by. To show the practical feasibility of our approach, we integrated it into the kernel of the open source database system PostgreSQL.

To get results that remain valid as time passes by, we keep ongoing time points uninstantiated during query processing. We do so by, conceptually, evaluating a query at every possible reference time and then combining these query results into a single, ongoing query result for all reference times. To represent this query result, we propose *ongoing relations* that associate each tuple with a reference time attribute RT. The value of the RT attribute includes the reference times when *now* can be instantiated in the tuple and the tuple belongs to the instantiated relations. The RT value of a tuple accommodates selecting tuples at some reference times only and is restricted by predicates and functions on ongoing attributes.

The approach supports evaluating the standard functions for time intervals, i.e., intersection, difference, and union to results that remain valid. At each reference time, the function result consists of the expected time intervals. We represent the function results as a combination of ongoing intervals and the reference time when the ongoing interval is part of the result. To

store the function results in relations, we leverage ongoing relations with a single reference time attribute that integrates the restrictions from the results of all interval functions.

Aggregating data is an important and frequently performed task in database systems to summarize data. Our approach proposes an aggregation operator for ongoing relations that correctly and efficiently groups and aggregates ongoing tuples. The aggregation result is an ongoing relation that remains valid as time passes by. We integrated our aggregation algorithm into the query processing pipeline of the PostgreSQL database system. Conceptually, the algorithm first divides the input tuples into fixed groups, each containing the tuples with equal fixed grouping attribute values, then further divides each fixed group into ongoing groups according to the ongoing grouping attributes and the reference time attribute RT, and finally aggregates each ongoing group into a single result tuple with the aggregate functions. The aggregation algorithm leverages existing, optimized strategies of the database system to determine the fixed groups and to calculate the aggregate values. Our incremental calculation of the fixed groups, the ongoing groups, and the aggregate values avoids materializing the tuples that belong to an aggregation group and keeps the memory consumption per ongoing group constant, independent of the size of the group.

Future Work: In this thesis, we focus on ongoing time points and time intervals as the ongoing values in a relation. Functions like the duration of ongoing time intervals require new ongoing data types like *ongoing integers*. The challenge is to find a finite representation of ongoing integers, such that it supports the standard functions on integers, including addition, subtraction, and multiplication. With ongoing integers as a new data type, it would be interesting to extend our aggregation operator on ongoing relations to support aggregate functions on ongoing values as well.

Currently, we use basic evaluation strategies for the relational algebra operators on ongoing relations. To improve the performance of our operators, we plan to enable optimization for them. A first step is to develop index access methods for ongoing time points, based on the approaches for indexing fixed time intervals, and discuss query classes that benefit from these indexes.

The focus of this thesis is the storage of ongoing data types and query processing. Another important functionality of database systems is to preserve the integrity of the data. We want to define integrity constraints, including primary key and foreign key constraints, for ongoing relations, such that the constraints remain satisfied as time passes by and can be checked efficiently.

Bibliography

[ABPT01] Mikkel Agesen, Michael H. Böhlen, Lasse O. Poulsen, and Kristian Torp. A Split Operator for Now-Relative Bitemporal Databases. In *Proceedings 17th International Conference on Data Engineering (ICDE)*, pages 41–50, 2001.

[APS+16] Luca Anselma, Luca Piovesan, Abdul Sattar, Bela Stantic, and Paolo Terenziani. A Comprehensive Approach to Now in Temporal Relational Databases: Semantics and Representation. *IEEE Transactions on Knowledge and Data Engineering*, 28(10):2538–2551, 2016.

[APT16] Luca Anselma, Luca Piovesan, and Paolo Terenziani. A 1NF Temporal Relational Model and Algebra Coping with Valid-time Temporal Indeterminacy. *Journal of Intelligent Information Systems*, 47(3):345–374, 2016.

[ASTS13] Luca Anselma, Bela Stantic, Paolo Terenziani, and Abdul Sattar. Querying Now-Relative Data. *Journal of Intelligent Information Systems*, 41(2):285–311, 2013.

[BBJS97] John Bair, Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. Notions of Upward Compatibility of Temporal Query Languages. *Wirtschaftsinformatik*, 39(1):25–34, 1997.

[BGJ06] Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. Multi-Dimensional Aggregation for Temporal Data. In *Proceedings of the 10th International Conference on Advances in Database Technology*, EDBT, pages 257–275, 2006.

[BJ02] Michael H. Böhlen and Christian S. Jensen. Temporal Data Model and Query Language Concepts. In Hossein Bidgoli, editor, *Encyclopedia of Information Systems*, pages 437–453. Academic Press, 2002.

[BJ09] Michael H. Böhlen and Christian S. Jensen. Sequenced Semantics. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 2619–2621. Springer, 2009.

[BJS00] Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. Temporal Statement Modifiers. *ACM Transactions on Database Systems (TODS)*, 25(4):407–456, 2000.

[BM17] Panagiotis Bouros and Nikos Mamoulis. A Forward Scan Based Plane Sweep Algorithm for Parallel Interval Joins. *Proceedings of the VLDB Endowment*, 10(11):1346–1357, 2017.

[CB17] Francesco Cafagna and Michael H. Böhlen. Disjoint Interval Partitioning. *The VLDB Journal*, 26(3):447–466, 2017.

[CDI+97] James Clifford, Curtis Dyreson, Tomás Isakowitz, Christian S. Jensen, and Richard T. Snodgrass. On the Semantics of Now in Databases. *ACM Transactions on Database Systems (TODS)*, 22(2):171–214, 1997.

[COTN08] Su Chen, Beng Chin Ooi, Kian-Lee Tan, and Mario A. Nascimento. ST2B-tree: A Self-Tunable Spatio-Temporal B+-tree Index for Moving Objects. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 29–42. ACM, 2008.

[DBG14] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Overlap Interval Partition Join. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1459–1470. ACM, 2014.

[DBGJ16] Anton Dignös, Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. Extending the Kernel of a Relational DBMS with Comprehensive Support for Se-

quenced Temporal Queries. *ACM Transactions on Database Systems (TODS)*, 41(4):26:1–26:46, 2016.

[DDL02] Christopher John Date, Hugh Darwen, and Nikos Lorentzos. *Temporal Data & The Relational Model*. Elsevier, 2002.

[DGN+19] Anton Dignös, Boris Glavic, Xing Niu, Johann Gamper, and Michael H. Böhlen. Snapshot Semantics for Temporal Multiset Relations. *Proceedings of the VLDB Endowment*, 12(6):639–652, 2019.

[DJTS09] Curtis E. Dyreson, Christian S. Jensen, and Richard T. Snodgrass. Now in Temporal Databases. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 1920–1924. Springer, 2009.

[FM96] Marcelo Finger and Peter McBrien. On the Semantics of Current-Time in Temporal Databases. In *Proceedings of the 11th Brazilian Symposium on Databases*, pages 324–337, 1996.

[GS05] Ralf Hartmut Güting and Markus Schneider. *Moving Objects Databases*. Elsevier, 2005.

[GSSY98] Jose Alvin G. Gendrano, Rachana R. Shah, Richard T. Snodgrass, and Jian Yang. University Information System (UIS) Dataset. TimeCenter CD-1, 1998.

[HXL05] Haibo Hu, Jianliang Xu, and Dik Lun Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 479–490. ACM, 2005.

[JCG+92] Christian S. Jensen, James Clifford, Shashi K. Gadia, Arie Segev, and Richard T. Snodgrass. A Glossary of Temporal Database Concepts. *SIGMOD Record*, 21(3):35–43, 1992.

[JL01] Christian S. Jensen and David B. Lomet. Transaction Timestamping in (Temporal) Databases. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB, pages 441–450, 2001.

[JS09] Christian S. Jensen and Richard T. Snodgrass. Valid-Time and Transaction-Time Relation. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 3254–3254. Springer, 2009.

[KM12] Krishna Kulkarni and Jan-Eike Michels. Temporal Features in SQL:2011. *SIGMOD Record*, 41(3):34–43, 2012.

[LLBY14] Jongtae Lim, Yoonjoon Lee, Kyoungsoo Bok, and Jaesoo Yoo. A Continuous Reverse Skyline Query Processing for Moving Objects. In *2014 International Conference on Big Data and Smart Computing (BIGCOMP)*, pages 66–71, 2014.

[LM97] Nikos A. Lorentzos and Yannis G. Mitsopoulos. SQL Extension for Interval Data. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):480–499, 1997.

[LPD13] Ahmed Lamkanfi, Javier Pérez, and Serge Demeyer. The Eclipse and Mozilla Defect Tracking Dataset: A Genuine Dataset for Mining Bug Information. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 203–206. IEEE Press, 2013.

[LSM05] Ines Fernando Vega Lopez, Richard T. Snodgrass, and Bongki Moon. Spatiotemporal Aggregate Computation: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):271–286, 2005.

[MB20a] Yvonne Mülle and Michael H. Böhlen. Functions on Ongoing Intervals. In *under review*, 2020.

[MB20b] Yvonne Mülle and Michael H. Böhlen. Query Results over Ongoing Databases that Remain Valid as Time Passes By. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1429–1440, 2020.

[MB20c] Yvonne Mülle and Michael H. Böhlen. Query Results over Ongoing Databases that Remain Valid as Time Passes By (Extended Version). *Technical Report CoRR*, 2020. URL: https://arxiv.org/pdf/2001.05722.pdf.

[MS93] Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, 1993. pages 69,107,459.

[NATM15] Sarana Nutanong, Mohammed Eunus Ali, Egemen Tanin, and Kyriakos Mouratidis. Dynamic Nearest Neighbor Queries in Euclidean Space. In Shashi Shekhar, Hui Xiong, and Xun Zhou, editors, *Encyclopedia of GIS*, pages 1–7. Springer, 2015.

[PHD16] Danila Piatov, Sven Helmer, and Anton Dignös. An Interval Join Optimized for Modern Hardware. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1098–1109, 2016.

[Pos20a] PostgreSQL. Range Functions and Operators, 2020. URL: http://www.postgresql.org/docs/9.4/functions-range.html.

[Pos20b] PostgreSQL. Range Types, 2020. URL: http://www.postgresql.org/docs/9.4/rangetypes.html.

[Pos20c] PostgreSQL. Set-Returning Functions, 2020. URL: https://www.postgresql.org/docs/9.4/functions-srf.html.

[Sno87] Richard T. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems (TODS)*, 12(2):247–298, 1987.

[SSJ94] Michael D. Soo, Richard T. Snodgrass, and Christian S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *Proceedings of the 10th International Conference on Data Engineering (ICDE)*, pages 282–292, 1994.

[SST09] Bela Stantic, Abdul Sattar, and Paolo Terenziani. The POINT Approach to Represent Now in Bitemporal Databases. *Journal of Intelligent Information Systems*, 32(3):297–323, 2009.

[STS03] Bela Stantic, John Thornton, and Abdul Sattar. A Novel Approach to Model NOW in Temporal Databases. In *Proceedings of the 10th International Symposium on Temporal Representation and Reasoning and Fourth International Conference on Temporal Logic*, pages 174–180, 2003.

[TCG+93] Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard T. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation*. Benjamin-Cummings Publishing Co., Inc., 1993.

[Ter20a] Teradata. Period Functions and Operators, 2020. URL: https://docs.teradata.com/reader/kmuOwjp1zEYg98JsB8fu_A/PTuowP0k01KJY0WZy7tAcQ.

[Ter20b] Teradata. Period (Timestamp) Data Type, 2020. URL: https://docs.teradata.com/reader/WurHmDcDf31smikPbo9Mcw/eYATDyMVK6doc253Xv28lQ.

[TJB97] Kristian Torp, Christian S. Jensen, and Michael H. Böhlen. Layered Temporal DBMS's: Concepts and Techniques. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 371–380, 1997.

[TJS00] Kristian Torp, Christian S. Jensen, and Richard T. Snodgrass. Effective Timestamping in Databases. *The VLDB Journal*, 8(3-4):267–288, 2000.

[TJS04] Kristian Torp, Christian S. Jensen, and Richard T. Snodgrass. Modification Semantics in Now-Relative Databases. *Information Systems*, 29(8):653–683, 2004.

[Tom96] David Toman. Point-Based vs. Interval-Based Temporal Query Languages. In *Proceedings of PODS*, pages 58–67, 1996.

[Tom98] David Toman. Point-Based Temporal Extensions of SQL And Their Efficient Implementation. In *Temporal Databases: Research and Practice*, pages 211–237. Springer, 1998.

[TP02] Yufei Tao and Dimitris Papadias. Time-Parameterized Queries in Spatio-Temporal Databases. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 334–345. ACM, 2002.

[TPC17] TPC. TPC Transaction Processing Performance Council, TPC-H, 2017. URL: http://www.tpc.org/tpch.

[TYJ09] Kostas Tzoumas, Man Lung Yiu, and Christian S. Jensen. Workload-Aware Indexing of Continuously Moving Objects. *Proceedings of the VLDB Endowment*, 2(1):1186–1197, 2009.

[ZQL+12] Rui Zhang, Jianzhong Qi, Dan Lin, Wei Wang, and Raymond Chi-Wing Wong. A Highly Optimized Algorithm for Continuous Intersection Join Queries over Moving Objects. *The VLDB Journal*, 21(4):561–586, 2012.